Inter-Office Memorandum

To	Cedar Interest	Date	January 20, 1981
From	Doug Wyatt	Location	Palo Alto
Subject	The Cedar Graphics Package	Organization	PARC/ISL

XEROX

DRAFT - DRAFT - DRAFT - DRAFT - DRAFT

Caution: this is still an incomplete draft. Several sketchy explanations need to be filled out, and a few omitted ones need to be filled in.

Filed on: [Ivy]<CedarGraphics>CedarGraphics.memo

The Graphics interface and the interfaces on which it depends can be found on [Ivy]<Cedar>Graphics>. The implementation is a part of Cedar.bcd.

Definitions files:

Graphics.bcd -- *the main interface to Cedar Graphics*
Vector.bcd -- *definitions for vectors; included by* Graphics
Cubic.bcd -- *definitions for cubic curves; included by* Graphics
Style.bcd -- *definitions for drawing style; included by* Graphics

Introduction

The Cedar Graphics package provides low level display facilities for Cedar. It is capable of driving a wide range of raster display devices while presenting a display-independent interface to its clients. Users of the package can specify abstract geometrical objects to be displayed (filled areas, lines, and text) without being concerned with the peculiar characteristics of the actual display device in use.

Cedar Graphics provides a powerful set of primitive operations for driving a virtual display. It is intended as a foundation for the construction of higher level display abstractions such as the Cedar Documents facilities. Cedar Documents is expected to be the primary client of Cedar Graphics. Most Cedar programmers will not deal directly with the Graphics interface, but will use the Document facilities that are built on top of it.

This memo discusses some of the design principles behind Cedar Graphics, then describes the current interface in detail. Future incarnations of the package will adhere to the same principles, but the interface has not reached its final form. Cedar Graphics developed as a Mesa program, apart from Cedar, and does not yet take advantage of Cedar-specific facilities. In particular, it still uses uncounted storage, and presents an unsafe interface. The performance of the current implementation leaves much room for improvement. A substantial revision of the package which fixes many of these problems is in progress; see CedarGraphicsRevision.memo (also in progress).

Principles and Concepts

Clients of the Graphics interface deal with an abstraction called a *display context*. A display context represents a virtual display; its primary function is to decouple the user from the characteristics of the underlying device. Display contexts also hold certain state information which affects the appearance of objects drawn in those contexts.

The devices supported may differ in a number of respects. Different displays usually differ in resolution and size. Every display context carries a linear transformation which maps from the coordinate system perceived by the user into the coordinate system required by the device. This transformation is initialized so that the user always sees the same initial coordinate system regardless of what device the display context is driving.

Devices may also differ in the amount of information used to represent each pixel. The graphics pipeline provides the device with a high precision description of each shape to be drawn; the device is supposed to produce the best rendition it can. A simple bitmap display may represent colors as different stipple patterns. A gray-level display may represent colors as different gray values and apply anti-aliasing techniques to create the appearance of smooth edges.

Cedar Graphics is a *low level* graphics package. It maintains no history of display actions apart from the dots on the display. Operations that alter the state of the display context do not affect the current picture on the display, but only items drawn subsequently. Think of the display context as a transducer. Clients are responsible for retaining graphical objects they want to display repeatedly. Several features are designed to facilitate the management of graphical objects.

The idea is that the client should have a procedure that takes a display context as a parameter and draws the appropriate figure in that context. In general, information is communicated by a series of procedure calls rather than the passing of large or complex data structures.

Description of device, world, and user coordinate systems.

Display positions are represented throughout the package as floating point numbers (type **REAL**). A wide range or coordinates can be accommodated, while high precision is maintained. The performance of Cedar Graphics relies heavily on an efficient implementation of operations on **REAL**s. Floating point operations are now implemented in microcode on both Dolphin and Dorado. Floating point hardware is being built for the Dolphin.

Note that the transformation to device coordinates does *not* entail any loss of precision. Though the *units* of measurement are those natural to the device (e.g., the pixel size), positions retain the full *precision* of floating point numbers. Truncation or rounding takes place only at the last moment, when scan conversion determines which pixels are to be changed.

Combinations of mapping and clipping operations allow a display context to represent some smaller region of the display, while preserving the client's illusion that it owns the entire screen. Several coexisting display contexts may drive disjoint portions of the display in this manner. This is the mechanism that the Documents package uses to implement windows.

Also note that the bulk of the Cedar Graphics implementation itself is device-independent. Device drivers are (will be) relatively small: they need only know how to scan convert simple convex polygons, and how to interpret colors (and how to set up the initial coordinate system and describe its limits).

State held by a display context:

> *Transformation*: current transformation matrix and stack
>
> *Position*: current position on the display. interpretation changes with transformation
>
> *Style*: painting function, texture, and font still too Alto-specific; need a uniform way to specify color; should always replace; think of characters as masks; textures are synthetic images

Clipper: current clipping region

Path: current path in progress and stack

Box: bounding box for a sequence of display actions

Interface Layers

Client interface: carefully hides the device characteristics. This is intended to be a relatively stable interface. Clients of this interface should be unaffected by the addition of new devices or changes in interfaces (such as the Pipe or Device interfaces) internal to the implementation.

Device interface: modules implementing different displays export this interface. The device only has to understand simple convex polygons and the encoding of color. In the current implementation, the device is also assumed to understand fonts; this is a bad idea, and will soon be changed.

Internal interfaces: may be useful for other graphics applications. Naturally, clients of these interfaces pay the price for a more intimate coupling to the implementation of the package. These relatively volatile interfaces are subject to change as new capabilities are added or performance tuning is done.

The Graphics Interface

The following is a detailed description of the current version of the Graphics interface. *Note: this is likely to change in the near future.*

Display Contexts

This section describes procedures for creating, copying, and destroying display contexts; and procedures for saving and restoring some of the context's state.

```
Graphics.DisplayContext: TYPE = Opaque.DisplayContext;

Opaque.DisplayContext: TYPE = LONG POINTER TO <opaque type>;
```

A **DisplayContext** denotes a particular display context. Most of the Graphics procedures take a **DisplayContext** as a parameter.

```
Graphics.DeviceHandle: TYPE = Opaque.DeviceHandle;

Opaque.DeviceHandle: TYPE = LONG POINTER TO <opaque type>;
```

A **DeviceHandle** denotes a particular display device. **DeviceHandle**s for non-standard devices are obtained through other interfaces which are not described here.

```
Graphics.NewContext: PROC[device: Graphics.DeviceHandle _ NIL]
    RETURNS[Graphics.DisplayContext];
```

This creates a new display context for the specified display device. If **device=NIL**, the context uses the standard Alto display. The initial state of the context is as follows:

Transformation: world coordinate system (see below).
Position: [0,0].
Style: **PaintingFunction** is **replace**; **Texture** is **black**; **Font** is undefined.
Clipper: the entire display.
Path: empty.
Box: disabled.

Cedar Graphics defines the world coordinate system as follows: the origin is at the bottom left corner of the display; increasing x moves rightward, and increasing y moves upward. The units of distance are *points*; there are 72 points to an inch. Points were chosen for convenience in specifying font sizes. Also, a pixel on the Alto display is defined to be 1 point square.

```
Graphics.PushContext: PROC[dc: Graphics.DisplayContext];

Graphics.PopContext: PROC[dc: Graphics.DisplayContext];
```

A display context maintains a stack which can save and restore some of its state: the transformation, position, style, line width, and font. **PushContext** pushes a snapshot of the current context state onto the stack. **PopContext** restores the context state to the state on the top of the stack, and pops the stack.

```
Graphics.CopyContext: PROC[dc: Graphics.DisplayContext]
    RETURNS[Graphics.DisplayContext];
```

This creates a new display context whose state is copied from the specified context.

```
Graphics.FreeContext: PROC[dcPtr: LONG POINTER TO Graphics.DisplayContext];
```

This *unsafe* operation frees the storage occupied by the specified display context. Naturally, this will go away when Cedar Graphics is converted to use Cedar's counted storage.

Drawing Shapes

```
Graphics.Vec: TYPE = Vector.Vec;

Vector.Vec: TYPE = RECORD[x,y: REAL];
```

A **Vec** usually represents the x and y coordinates of a location on the display. A few procedures take a **Vec** which represents a displacement from a location other than the origin; see the descriptions of the individual procedures for details. The **Vector** interface provides a collection of **INLINE** procedures for performing operations on **Vec**s.

```
Cubic.Coeffs: TYPE = RECORD[c0,c1,c2,c3: Vector.Vec];
```

These are the coefficients of a parametric cubic curve:

x = **c0.x** + **c1.x** t + **c2.x** t^2 + **c3.x** t^3
y = **c0.y** + **c1.y** t + **c2.y** t^2 + **c3.y** t^3

where the parameter t ranges from 0 to 1. Cedar will provide a spline package, which can generate a series of cubics representing a smooth curve passing through specified knots. Most clients will want to use this rather than computing coefficients themselves.

The following procedures are used to examine and alter the current position, and to draw straight lines of a given width.

```
Graphics.GetPosition: PROC[dc: Graphics.DisplayContext] RETURNS[Graphics.Vec];
```

This returns the current position in user coordinates. Note that a change in the current transformation will change the coordinates returned by **GetPosition**, even though the current position remains fixed on the display.

```
Graphics.MoveTo: PROC[dc: Graphics.DisplayContext, v: Graphics.Vec];
```

This changes the current position to **v**.

```
Graphics.RelMoveTo: PROC[dc: Graphics.DisplayContext, v: Graphics.Vec];
```

This moves the current position by the displacement **v**.

Graphics.DrawTo: PROC[dc: Graphics.DisplayContext, v: Graphics.Vec];

This draws a straight line beginning at the current position and ending at position **v**, using the current line width (see below). The current position is moved to the end of the line.

Graphics.RelDrawTo: PROC[dc: Graphics.DisplayContext, v: Graphics.Vec];

This draws a straight line beginning at the current position and ending at the current position plus **v**, using the current line width (see below). The current position is moved to the end of the line.

Graphics.SetLineWidth: PROC[dc: Graphics.DisplayContext, w: REAL];

This establishes the width of lines drawn by **DrawTo** or **RelDrawTo**. Note that **w** is interpreted in the *user* coordinate system; its effect is altered by changes in the current transformation. For example, if the coordinate system is scaled by 2, subsequent lines will be twice as thick.

Graphics.GetLineWidth: PROC[dc: Graphics.DisplayContext] RETURNS[REAL];

This returns the current line width.

Areas

The following procedures are for drawing filled areas. A shape to be filled is defined by a *path*. A path is a set of one or more polygonal boundaries; the boundaries may intersect themselves or each other.

Graphics.StartAreaPath: PROC[dc: Graphics.DisplayContext,
oddeven: BOOLEAN _ FALSE];

This begins a new path. The display context maintains a stack of pending paths. If an incomplete path is in progress when **StartAreaPath** is called, that path is pushed onto the stack. The **oddeven** parameter affects the treatment of self-intersecting areas. If **oddeven** is **FALSE**, points with *nonzero* wrap number are considered to be in the interior; if **oddeven** is **TRUE**, only points with *odd* wrap number are considered to be in the interior. A description of wrap number should go here.

Graphics.EnterPoint: PROC[dc: Graphics.DisplayContext, v: Graphics.Vec];

This enters the point **v** into the current path.

Graphics.EnterCubic: PROC[dc: Graphics.DisplayContext, c: POINTER TO Cubic.Coeffs];

This extends the current path with the specified curve segment.

Graphics.NewBoundary: PROC[dc: Graphics.DisplayContext];

This closes the current boundary, connecting the last point to the first point of the boundary. A subsequent **EnterPoint** will begin a new boundary.

Graphics.DestroyPath: PROC[dc: Graphics.DisplayContext];

This disposes of the current path, without drawing it, and pops the path stack.

Graphics.DrawArea: PROC[dc: Graphics.DisplayContext];

This draws the area defined by the current path, filling it using the current texture and painting function, and then pops the path stack.

Graphics.DrawRectangle: PROC[dc: Graphics.DisplayContext, ll,ur: Graphics.Vec];

This draws the specified rectangle (**ll** and **ur** are its lower left and upper right corners), filling it using the current texture and painting function. The path stack is unchanged.

```
Graphics.DrawScreenArea: PROC[dc: Graphics.DisplayContext];
```

This is equivalent to calling **StartAreaPath**, entering a path which surrounds the entire physical screen, then calling **DrawArea**. Since all areas are clipped, the effect of **DrawScreenArea** is to fill all of the current clipping region, using the current texture and painting function.

Paint

The following procedures determine the display context parameters used to fill lines, areas, and text.

```
Graphics.PaintingFunction: TYPE = Style.PaintingFunction;

Style.PaintingFunction: TYPE = {replace, paint, invert, erase};
```

These are equivalent to the BitBlt functions of the same names. This is much too Alto-specific. PaintingFunction should go away. Replace can be used all the time if characters are properly treated as masks.

```
Graphics.Texture: TYPE = Style.Texture;

Style.Texture: TYPE = CARDINAL;
```

The texture value is interpreted as a 4x4 bit square which tiles the plane. The constants **white** and **black** are **Texture**s defined in the interface. This is even more Alto-specific and awful; **Color** should be used instead. Bitmap devices may substitute textures for colors; clients that really want to express textures should use (synthetic) images.

```
Graphics.SetPaint: PROC[dc: Graphics.DisplayContext,
    p: Graphics.PaintingFunction];

Graphics.GetPaint: PROC[dc: Graphics.DisplayContext]
    RETURNS[Graphics.PaintingFunction];

Graphics.SetTexture: PROC[dc: Graphics.DisplayContext, t: Graphics.Texture];

Graphics.GetTexture: PROC[dc: Graphics.DisplayContext] RETURNS[Graphics.Texture];
```

Transformations

These are for redefining the virtual coordinate system. Remember that changing the transformation does not alter objects previously displayed.

```
Graphics.Translate: PROC[dc: Graphics.DisplayContext, v: Graphics.Vec];
```

This translates the origin of the coordinate system by **v**.

```
Graphics.Scale: PROC[dc: Graphics.DisplayContext, v: Graphics.Vec];
```

This scales the coordinate system by **[v.x,v.y]**.

```
Graphics.Rotate: PROC[dc: Graphics.DisplayContext, angle: REAL];

Vector.Matrix: TYPE = RECORD[a11,a12,a21,a22: REAL];

Graphics.Concatenate: PROC[dc: Graphics.DisplayContext, m: Vector.Matrix];
```

```
Graphics.Map: PROC[sdc,ddc: Graphics.DisplayContext, sp: Graphics.Vec]
    RETURNS[dp: Graphics.Vec];

Graphics.ScreenPoint: PROC[dc: Graphics.DisplayContext, v: Graphics.Vec]
    RETURNS[Graphics.Vec];
```

Text

These are for defining fonts, displaying text, and determining font metrics.

```
Graphics.FontId: TYPE[1];

Graphics.MakeFont: PROC[family: STRING, bold,italic: BOOLEAN _ FALSE]
    RETURNS[Graphics.FontId];

Graphics.SetFont: PROC[dc: Graphics.DisplayContext,
    font: Graphics.FontId, size: REAL];

Graphics.DisplayChar: PROC[dc: Graphics.DisplayContext, c: CHARACTER];

Graphics.DisplayString: PROC[dc: Graphics.DisplayContext, s: LONG STRING];

Graphics.CharData: TYPE = RECORD[size,origin,width: Graphics.Vec];

Graphics.GetCharBox: PROC[dc: Graphics.DisplayContext, c: CHARACTER,
    data: POINTER TO Graphics.CharData];

Graphics.GetStringBox: PROC[dc: Graphics.DisplayContext, s: LONG STRING,
    data: POINTER TO Graphics.CharData];

Graphics.GetFontBox: PROC[dc: Graphics.DisplayContext,
    data: POINTER TO Graphics.CharData];
```

Clipping and Bounding Boxes

Here is an introductory sentence.

```
Graphics.SetClipArea: PROC[dc: Graphics.DisplayContext];
```

This sets the current clipping region to the area defined by the current path. The path is clipped to the physical display boundaries, but the current clipper is ignored.

```
Graphics.IntersectClipArea: PROC[dc: Graphics.DisplayContext];
```

This is similar to **SetClipArea**, but it clips the path with the current clipper. In other words, the new clipping region will be the area that would have been drawn if you had called **DrawArea** instead of **SetClipArea**. This should become the default, and SetClipArea should go away. The recipient of a display context should not be allowed to reach outside its clipping region!

Bounding boxes may be used to perform quick clipping tests for complex objects. Explanation to come.

```
Graphics.BoundingBox: TYPE = ARRAY [0..4) OF Graphics.Vec;

Graphics.InitBoxer: PROC[dc: Graphics.DisplayContext];

Graphics.StopBoxer: PROC[dc: Graphics.DisplayContext,
    bbox: POINTER TO Graphics.BoundingBox];
```

```
Graphics.PushClipBox: PROC[dc: Graphics.DisplayContext,
    bbox: POINTER TO Graphics.BoundingBox];

Graphics.PopClipBox: PROC[dc: Graphics.DisplayContext];

Graphics.Visible: PROC[dc: Graphics.DisplayContext] RETURNS[BOOLEAN];
```

Images and other loose ends

Here is an introductory sentence.

```
Graphics.DrawImage: PROC[dc: Graphics.DisplayContext,
    image: Graphics.ImageHandle];
```

M*ore discussion*.