Unicon's OpenGL 2D and Integrated 2D/3D Graphics Implementation

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Kevin Z. Young

Major Professor: Clinton Jeffery, Ph.D.

Committee Members: Terence Soule, Ph.D.; Robert Heckendorn, Ph.D.

Department Administrator: Terence Soule, Ph.D.

August 2020

Authorization to Submit Thesis

This thesis of Kevin Z. Young, submitted for the degree of Master of Science with a Major in Computer Science and titled "Unicon's OpenGL 2D and Integrated 2D/3D Graphics Implementation," has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor:	______________________ Clinton Jeffery, Ph.D.	____________ Date
Committee Members:	______________________ Terence Soule, Ph.D.	____________ Date
	______________________ Robert Heckendorn, Ph.D.	____________ Date
Department Chair:	______________________ Terence Soule, Ph.D.	____________ Date

Abstract

Writing an OpenGL implementation for Unicon's 2D facilities introduces an opportunity to create an integrated 2D/3D graphics mode that is intuitive to the end user. The implementation must be backwards-compatible with existing Unicon 2D graphics programs. The completion of the project will result in the release of the OpenGL implementation in public beta. Evaluation of this implementation shows it be qualitatively and quantitatively robust.

Acknowledgments

I would like to acknowledge my parents for giving me to opportunity to pursue higher education. I would also like to acknowledge my major professor, Dr. Clinton Jeffery for all of the support and advice given to me through this process and my committee members for taking interest and reviewing my work. This thesis was supported in part by the National Library of Medicine.

Table of Contents

Authorization to Submit Thesis ii
Abstract iii
Acknowledgments iv
Table of Contents v
List of Tables viii
List of Figures ix
List of Code Listings xi
List of Acronyms xiii
Chapter 1: Introduction 1
Chapter 2: Related Work 4
Chapter 3: Design 6
  Why OpenGL 6
  Backing Store vs Display List 7
  2D Facilities 8
  Integrated 2D/3D Facilities 10
    User Interface 11
    Render Semantics 11
Chapter 4: Implementation 13
  2D Facilities 13
    Windowing 14
    Render Context 18
    Display List 20
    Textures 23
    Stencil Buffer 25
    Context Attributes 27

Graphical Primitives 42
Integrated 2D/3D Facilities 60
Viewing Volume Sectioning 60
Chapter 5: Testing and Evaluation 65
Qualitative Evaluation 65
gpxtest 66
UI 68
IVIB 69
speedtest 69
jeb1 71
Quantitative Evaluation 72
speedtest 72
animation 75
Chapter 6: Conclusion 78
Release 78
Future Work 78
References 80
Appendix A: Developer Reference 82
Internal OpenGL 2D API 82
Core API 83
Platform-specific API 86
Display List Objects 88
Appendix B: User Guide 93
2D Features 93
Transparency 93
Animation with a Display List 94
Optimizing Performance 96

Integrated 2D/3D Features 96
Modifiable Display List Record Fields 97
Appendix C: Unicon Test Programs 102
speedtest.icn 102
animation.icn 105

List of Tables

5.1 Testing machine hardware specifications 65

5.2 Xlib vs OpenGL performance: Avg. of 5, 100,000 primitives (Machine 1) . . . 74

5.3 Xlib vs OpenGL performance: Avg. of 5, 100,000 primitives (Machine 2) . . . 75

5.4 Xlib vs OpenGL avg. performance: Avg. of 5, 100,000 primitives 76

5.5 Xlib vs OpenGL avg. perf.: Avg. of 5, 100,000 primitives (no **line/fillstyle**) . 76

5.6 OpenGL animation performance (Machine 1) 76

5.7 OpenGL animation performance (Machine 2) 76

List of Figures

1.1 A simple Unicon 2D graphics program 1

3.1 Unicon window and display list design diagram 9
3.2 Orthogonal viewing volume 12
3.3 Perspective viewing volume 12

4.1 UML diagram of the OpenGL implementation of Unicon windows 14
4.2 Unicon to OpenGL coordinate transformation 64

5.1 gpxtest (Xlib) 66
5.2 gpxtest (Mesa) 66
5.3 gpxtest dialog (Xlib) 66
5.4 gpxtest dialog (Mesa) 66
5.5 gpxtest (Mesa) 67
5.6 gpxtest (Nvidia) 67
5.7 gpxtest dialog (Mesa) 67
5.8 gpxtest dialog (Nvidia) 67
5.9 UI file selection (Xlib) 68
5.10 UI file selection (Mesa) 68
5.11 UI (Xlib) 68
5.12 UI (Mesa) 68
5.13 UI file selection (Mesa) 69
5.14 UI file selection (Nvidia) 69
5.15 IVIB: loading a layout (Xlib) 70
5.16 IVIB: loading a layout (Mesa) 70
5.17 speedtest: masked star polygons (Xlib) 70

5.18 speedtest: masked star polygons (Mesa) 70
5.19 Legacy jeb1 demo (Xlib) 71
5.20 Legacy jeb1 demo (Mesa) 71
5.21 Integrated 2D/3D jeb1 demo 72
5.22 Integrated 2D/3D jeb1 demo rendering artifact 72

List of Code Listings

1.1 Source code for a simple Unicon 2D graphics program . . . 1
4.1 Macros for managing and maintaining the current context . . . 15
4.2 GLX fields in internal C structures . . . 16
4.3 Algorithm for polling flush . . . 17
4.4 Macro wrapping for updating the Unicon render context . . . 18
4.5 Internal canvas render context fields . . . 19
4.6 Code for display list traversal . . . 21
4.7 Fields in internal display structure for storing OpenGL textures . . . 23
4.8 Internal procedures for texture allocation . . . 24
4.9 Internal function for texture deallocation . . . 25
4.10 The 3 bitplanes defined for Unicon's 2D graphics facilities . . . 26
4.11 Macros for controlling writes to the stencil buffer . . . 26
4.12 Macros for applying the stencil buffer . . . 27
4.13 OpenGL color struct . . . 27
4.14 OpenGL mutable color fields . . . 28
4.15 New foreground/background context fields . . . 28
4.16 The macro responsible for setting the OpenGL color state . . . 29
4.17 Macros for encoding and decoding **gamma** . . . 30
4.18 Get macros for struct color . . . 31
4.19 The macro for encoding **gamma** and setting the OpenGL color state . . . 31
4.20 The macro for applying **drawop** to the OpenGL color state . . . 32
4.21 OpenGL implementation of Unicon font macros . . . 33
4.22 Selective vertex sampling to segment circles and arcs . . . 33
4.23 The internal function for segmenting straight line segments . . . 34
4.24 Implementation of attribute **linestyle** . . . 35

4.25 Implementation of attribute **fillstyle** 36
4.26 The internal C function that renders **pattern** to the pattern bitplane 37
4.27 The implementation of clipping 40
4.28 Macros for removing and adding **dx** and **dy** 41
4.29 The base geometric rendering macro 42
4.30 Internal C function for rendering geometric primitives 43
4.31 Macros for rendering image primitives 45
4.32 Macro wrapper for OpenGL texture loading function 47
4.33 Internal C function for rendering bi-level images 48
4.34 Internal C function for rendering String Images 50
4.35 Internal C function for rendering images read from files 51
4.36 Fields added to the internal font structure 53
4.37 Internal C function for rendering font glyphs 54
4.38 Internal C function for copying rectangular areas 56
4.39 Internal C function for erasing rectangular areas 59
4.40 Viewing volume related macros 61
4.41 Macros for apply rendermode changes 62
4.42 The algorithm for rendering 2D and 3D display lists 62
4.43 Coordinate transformation macros 64
B.1 Obtaining the reference to a display list record 94
B.2 Changing the position of a primitive using **x** and **y** fields 94
B.3 Changing the position of primitives using **Dx** and **Dy** 95
B.4 Simple animation loop program 95
B.5 Rendermode switching with one canvas and one context 97
B.6 Rendermode switching with one canvas and two contexts 97
C.1 Source code for **speedtest.icn** 102
C.2 Source code for **animation.icn** 105

List of Acronyms

API application programming interface

GUI graphical user interface

HUD heads-up display

US unsigned short

UC unsigned char

GLF float

Chapter 1: Introduction

Unicon is a very-high level, goal-directed, object oriented, general purpose applications language with built-in graphics facilities designed to be intuitive for the end user [11]. Unicon's 2D graphics facilities were written for each of the three main platforms (Linux, Windows, and macOS) using platform specific graphics application programming interfaces (APIs). The current Linux and macOS implementations use X11 while the Windows implementation uses the native API. See Listing 1.1 for a simple Unicon 2D graphics program and Figure 1.1 for the output.

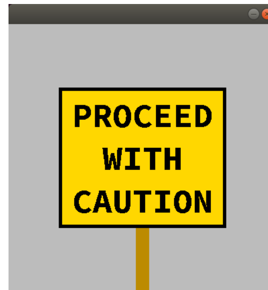

Figure 1.1: A simple Unicon 2D graphics program

```
link graphics
$include "keysyms.icn"

procedure main()
   &window := open("", "g", "size=400,400","fg=black")

   FillRectangle(75,95,250,210)
   Fg("gold")
   FillRectangle(80,100,240,200)
   Fg("dark brown")
   FillRectangle(190,305,20,95)
   Fg("black")
   Font("mono,50,bold")
   leading := WAttrib("leading")
   starty := 100+(200-3*leading)/2
   DrawString(200-TextWidth("PROCEED")/2,starty+leading,"PROCEED")
   DrawString(200-TextWidth("WITH")/2,starty+2*leading,"WITH")
   DrawString(200-TextWidth("CAUTION")/2,starty+3*leading,"CAUTION")

   repeat {
      case Event() of {
         "q": exit(0)
         }
      }
end
```

Listing 1.1: Source code for a simple Unicon 2D graphics program

A significant advantage of writing a cross-platform 2D graphics port is the reduction of code maintenance due to a shared code base. It also reduces the need to learn several different graphics APIs at a low level. This project strives to increase the portability of Unicon's graphics facilities by porting the legacy 2D facilities to OpenGL, an open-source, cross-platform graphics API, and extending the 3D facilities to include semantics for 2D drawing operations.

A by-product of porting the 2D facilities to OpenGL is the prospect of creating a new graphics mode. Unicon has both 2D and 3D graphics facilities, which have historically been independent of one another. Unicon currently provides subwindows as its only method to allow 2D and 3D windows to co-exist. This process is achieved by opening a 2D window and creating a 3D subwindow within it. Since the 3D facilities are written in OpenGL, porting the 2D facilities to OpenGL presents the opportunity to implement an integrated graphics environment in Unicon's runtime system which could provide a simpler, more intuitive method of rendering 2D and 3D graphics for the end user.

This thesis aims to not only implement an OpenGL port of the Unicon 2D graphics facilities, but also to pose the following questions to evaluate the end-product. Is the OpenGL implementation of Unicon's 2D graphics facilities feature-complete with respect to the legacy (bitmapped raster graphics) implementation? Can the OpenGL implementation's performance speed be made comparable and competitive to that of the legacy implementation? Will the proposed 2D/3D integrated facilities be practical for replacing the legacy subwindow-based applications? And finally, can the display list architecture of the OpenGL implementation be made to scale well enough for complex graphical user interfaces (GUIs) to be practical?

The evaluation of these research questions will determine the fate of OpenGL implementation of Unicon's 2D graphics facilities. If the new implementation performs with flying colors, then it is possible that it will replace the legacy facilities. If the new implementation misses all of the marks, then it will become dead code.

This thesis consists of the following chapters. Chapter 2 compares and contrasts related works to this project. Chapter 3 documents the design, implementation and semantics of the OpenGL 2D graphics port and how it impacts the end user experience. Chapter 4 details the implementation and describes the semantics for the new integrated 2D/3D graphics mode. Chapter 5 tests and evaluates the accuracy, efficacy, and usability of the OpenGL 2D port on a suite of standard Unicon 2D graphics programs. Chapter 6 presents a conclusion for the thesis and suggests future work. Appendix A provides a reference for Unicon developers. Appendix B supplies documentation for the end user on how to use the new features provided by the OpenGL port. Appendix C provides source code for Unicon test programs written for this thesis.

Chapter 2: Related Work

There are many APIs to choose from in computer graphics. These range from low level APIs which support bindings for multiple programming languages to high level graphics libraries built from low level APIs to facilitate ease of use. For programming languages, it is rare to have built-in graphics facilities. Instead, graphics are usually offered in the form of (optional) libraries. Unicon is one of the few programming languages to offer built-in graphics facilities. Many graphics libraries are geared towards either game or graphical user interface (GUI) development, rather than being general in purpose. While myriads of graphic libraries exist for every major programming language, this discussion of related works is confined to 2D and integrated 2D/3D graphics facilities in very high level languages built on top of OpenGL. Unicon's 2D graphics APIcontains 71 procedures, not including turtle graphics (16 procedures) and VIB (6 procedures) [10].

The main graphics library for Python—another very high-level object-oriented programming language—is tkinter, an object-oriented wrapper around the Tk GUI toolkit [8]. The Tk GUI toolkit was written for the scripting language Tcl, but has since developed bindings for many dynamic languages, Python included [3]. Tk is implemented using X11 [16], like Unicon's legacy 2D facilities. Tkinter has 19 widgets (classes) with 195 methods (91 of which are shared) [15].

Although tkinter is simple to learn, John Zelle wrote an even simpler graphics library around it, **graphics.py** as an introduction to graphics programming [18]. These libraries are designed for GUI programming which are comparable to Unicon's GUI class libraries, but do not handle any 3D rendering. **graphics.py** has 10 classes, with 54 methods total (8 of which are shared methods) [18].

Gosu is an object-oriented, easy-to-use 2D game development library for Ruby and C++ built on top of OpenGL [2]. Gosu offers window I/O, sounds, and music in addition to 2D graphics rendering. Ruby is the more natural binding for this graphics library due

to its dynamic nature, and the Gosu Ruby binding is more similar to Unicon's 2D graphics facilities as a result. Gosu has a different number of classes depending on the language binding, but appears to have an API of similar size to Unicon's 2D facilities. However, the main aspect that sets Gosu's Ruby binding apart from this thesis is library's lack of 3D graphics. Gosu's Ruby binding has 10 classes with a total of 63 methods [1].

Raylib is another easy-to-use game development library built on top of OpenGL which supports multiple language bindings, including C#, Go, Python and Ruby [6]. Where this library diverges from Unicon's is that it is built primarily for game development, rather than general graphics applications. Furthermore, raylib's API is lower level than Unicon's in that it loosely models OpenGL function calls and vector size types. Raylib has seven modules (core, textures, text, shapes, models, shaders, and audio) containing a total of 439 functions [6]. There are functions for specifying 2D and 3D rendering, similar to those implemented in Unicon with this thesis. Raylib appears to be the closest comparison to this thesis with the exception that it has a lower level of abstraction and its API size is larger.

SFML is a graphics library implemented with OpenGL for multimedia and 2D game development that supports multiple language bindings, including C++, Java, Ruby, Python, and Go [7]. This library includes five modules (audio, graphics, networks, systems, and windows), each with their sets of classes and methods [7]. The graphics module alone contains 25 classes with a total of 522 public methods [7].

Out of the plethora of graphics libraries built on top of OpenGL, a few are similar but not identical in scope and purpose to this thesis. Python's tkinter and Gosu's Ruby binding come closest in comparison to Unicon's 2D graphics facilities in terms of ease of use and size of API. Raylib is a contender for comparison for the overall scope of this thesis due to its 2D/3D graphics integration. However, Raylib's level of abstraction is lower than Unicon's, which is reflected in Raylib's much larger API.

Chapter 3: Design

Porting the Unicon 2D graphics facilities introduces important design questions. What underlying graphics API should be used? What rendering architecture would give the best trade-off between performance and ease-of-use? While maintaining backwards compatibility, are there new features or semantics that could be added to enhance usability?

3.1 Why OpenGL

The first of these questions was answered in the introduction. OpenGL was chosen because it is open-source (like Unicon), cross-platform, well supported by Khronos, and widely used in the field of graphics rendering. However, one issue that has not been touched on is which version of OpenGL should be used. As of the time this thesis is being written, the newest version of OpenGL is 4.6. Version 2.0 introduced the OpenGL Shading Language, which gives users more control by allowing them to bypass the fixed graphics pipeline using vertex shaders [4]. Many features related to OpenGL versions 1.x and the fixed graphics pipeline were deprecated in version 3.0 and removed in version 3.1.

Porting Unicon's 2D graphics facilities to OpenGL opens up the option of further porting to OpenGL ES, a subset of OpenGL for embedded systems and mobile devices. Android version 1.0 started supporting OpenGL ES versions 1.0 and 1.1 while Android versions 2.2 and 4.3 started supporting OpenGL ES versions 2.0 and 3.0 respectively [4]. OpenGL ES versions 1.x were based off of the OpenGL 1.5 specification while OpenGL ES version 2.0 was based off of the OpenGL 2.0 specification [5]. However, OpenGL ES 2.0 removes the fixed graphics pipeline entirely [4], while the fixed graphics pipeline is still an option in OpenGL versions 2.x.

The 3D facilities were written with OpenGL 1.2 with the goal of targeting the largest possible subset of computers able to run it [12]. Consequently, the 2D facilities were also written with accordance to OpenGL versions 1.x and the fixed graphics pipeline. Keeping

with the goal of device inclusion, version 1.x of OpenGL ES should be used for the future mobile device port. Therefore, Unicon should use OpenGL version 1.5 to facilitate a future port to Android and to be compatible with the 3D graphics facilities.

Another avenue for future work other than porting Unicon graphics to Android is to update the OpenGL version to 2.x and refactor everything to use vertex shaders. The OpenGL fixed graphics pipeline offers convenience and ease of use, but vertex shaders could offer a performance edge.

3.2 Backing Store vs Display List

Graphics rendering has two architectures: the backing store and the display list. Both architectures employ double-buffering, but the difference is that the backing store uses software double-buffering while the display list uses hardware double-buffering. Most legacy graphics APIs (raster graphics), like X, use the backing store architecture while newer vector graphics APIs use a display list architecture. Vector graphics makes use of both hardware acceleration and hardware double-buffering, which gives vector graphics an edge over raster graphics for non-bitmap related rendering. Bitmapped raster graphics excels at its namesake: bitmaps and pixel-based operations. OpenGL uses vector graphics, so it would slow down OpenGL rendering by using a backing store architecture because it takes more time to copy a buffer from the CPU to GPU than to use a switch implemented in hardware.

The main benefit of the display list for Unicon's graphics facilities is the flexibility it offers. The end user has the ability to inspect and modify the display list and the contents of each display list object, e.g. position, dimensions, font, text, etc. See Appendix B for specifics.

The main drawback of the display list is that each redrawing requires a list traversal of $O(n)$ for n items on the display list. However, Unicon is a high-level programming language striving for ease-of-use in its facilities. The display list allows users to only draw

a scene once and gives them the option of modifying the list for straightforward animation. In addition, Unicon's 3D facilities utilize a display list, and it is desirable to provide a consistent feel for the integrated 2D/3D graphics mode. Furthermore, implementing a display list in the 2D facilities caters to Unicon's core tenet of intuitive programming. To mitigate the potential slowdown from having a display list with tens-of-thousands of items or more, see Appendix B.1.3. Performance is discussed in Chapter 5.

3.3 2D Facilities

A primary requirement of the project is backwards compatibility. The port should not modify the semantics of the Unicon 2D graphics API in any way that would break existing 2D graphics programs. As long as legacy semantics are preserved, there is freedom to make the 2D facilities more Unicon-ish and give users semantics that will streamline code. The design choice of implementing a display list instead of a backing store exemplifies this.

The objects that are present on the 2D display list can be broken into two sets: graphical primitives and context attributes primitves. The graphical primitives represent the drawing operations of the Unicon 2D API, e.g. lines, points, rectangles, circles, arcs, etc. The context attribute primitives represent how the drawing can occur, e.g. colors, font, drawing style, clipping, and translation.

In Unicon, a window is a coupling between a *canvas* and a *context*. The *canvas* represents a drawable portion of the screen while the *context* determines how drawing operations are performed. A window binds a single canvas to a single context. However, references to canvases and contexts can be shared by multiple windows. Consequently, for one canvas and n contexts, there can be up to n unique window bindings with n different contexts drawing to a single canvas. The other extreme would be for n canvases and one context, there can be up to n unique window bindings where one context is bound to n canvases.

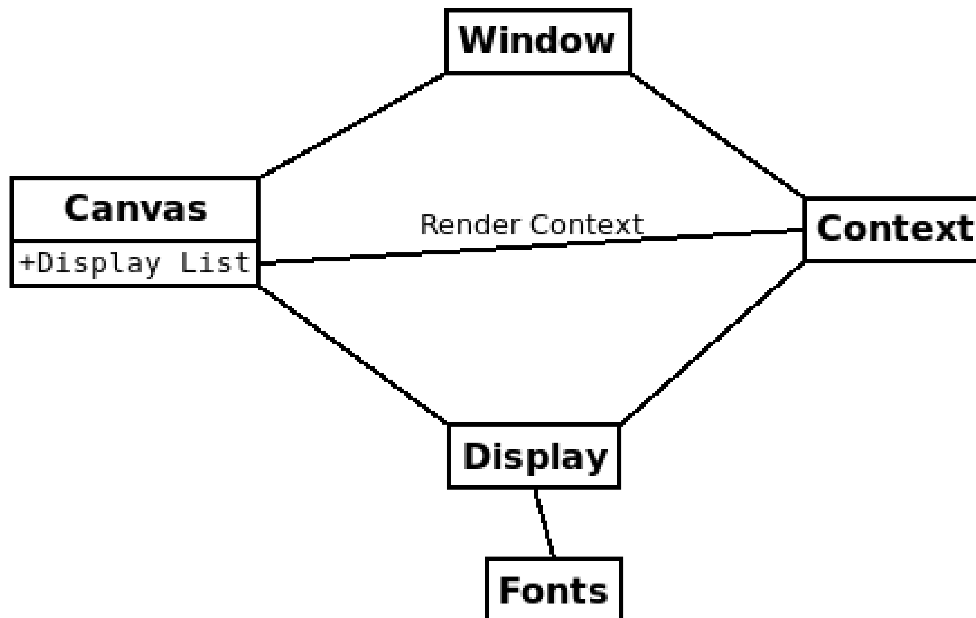

Figure 3.1: Unicon window and display list design diagram

In this scenario, the display list should be a part of the canvas, since it is a representation of what is drawn to the display. Because there may be multiple contexts that affect the rendering of graphical primitives, context attributes must be placed on the display list to preserve the order of operations to ensure proper rendering. To preserve the attributes of unique contexts, each canvas will own a *render context* in tandem with its display list. The semantics of the display list are as follows:

1. The display list is part of the canvas and has an accompanying render context.

2. A *render context* is a context that maintains the correct context attributes solely for the display list. The render context is initialized to the values of the first context its canvas is bound to.

3. The display list contains graphical and context attribute primitives.

4. A graphical primitive is appended to the display list when the corresponding Unicon 2D graphics library procedure is called. Graphical primitives are affected by render context attributes.

5. A context attribute primitive is appended to the display list whenever a context attribute is assigned a value. Context attribute primitives modify the render context.

6. The display list is traversed from front to back during a redraw operation to preserve order of operations. The render context is restored to its initial state before traversal and then context attribute primitives on the display list modify the render context's state accordingly.

7. Each canvas keeps track of the previously used context.

8. Prior to the addition of a display list object, if a new context is detected by the canvas, then context attribute primitives are appended to the display list for each differing attribute between the previous and new context.

Due to the order-preserving nature of the display list, it is possible to modify the coordinate fields of a graphical primitive and invoke a redraw to move the primitive. To move multiple, related graphical primitives—which are a sublist of the display list—one could add context attribute display list objects for translation, dx and dy, before such a sublist and modify those translational item fields before invoking a redraw. It is also possible to modify RGBA values of a color display list object to rapidly change colors of a sublist of graphical primitives on the display list. See Appendix B.1 for details about new features available to the 2D graphics facilities.

3.4 Integrated 2D/3D Facilities

The new integrated 2D/3D graphics mode is meant to be intuitive to Unicon users familiar with both the 2D and 3D facilities. Specifically, this integrated mode is introduced as an extension to the current 3D mode. Users may choose to use or ignore the new functionality.

3.4.1 User Interface

In the integrated graphics mode, there is a concept of a *render mode*. The render mode describes the state of graphics rendering and is mutually exclusive to 2D or 3D. Due to the integrated graphics mode being an extension of the 3D facilities, the default mode upon opening a Unicon integrated graphics window is 3D. A new context attribute **"rendermode"** was added to implement the switching mechanism between 2D and 3D render modes (see Section 4.2.1 for details). The user may then use the API for the corresponding current active render mode.

Some Unicon graphics procedures are mode-specific (**ReadImage()** for 2D and **Eye()** for 3D to name a couple) while others are shared between the two, e.g. **WAttrib()**, **Fg()**, **CopyArea()**, **EraseArea()**, **FillPolygon()**, **DrawPolygon()**, **DrawLine()**, **DrawPoint()**, and **DrawSegment()**). Any attempt to use an mode-specific procedure from the incorrect render mode will result in a runtime error, e.g. trying to render a 3D primitive while in 2D mode and vice versa. This design choice is to keep Unicon code more readable and less prone to surprises that may occur from implicit render mode switching.

3.4.2 Render Semantics

To envision how rendering works in this integrated environment, imagine there is a camera in 3D euclidian space which displays the world on the screen of the computer. The displayed objects are said to be within the viewing volume of the camera. This viewing volume can be of two different shapes: orthogonal (Figure 3.2) or perspective (Figure 3.3). An orthogonal viewing volume shows objects with one-to-one scaling, whereas the perspective viewing volume scales objects closer to the camera to look bigger and objects farther away from the camera to look smaller. Either way, the nearest objects that can be seen are on the near plane of the viewing volume, or on the same plane as the computer screen, and the furthest objects lie close to or on the far plane of the viewing volume. The 3D facilities use a perspective viewing volume.

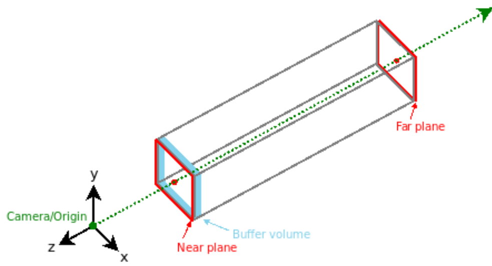

Figure 3.2: Orthogonal viewing volume

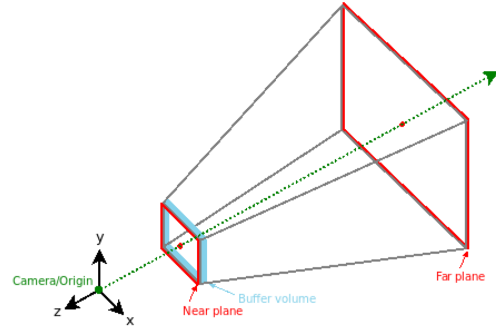

Figure 3.3: Perspective viewing volume

For this mode, we will reserve the near plane of the viewing volume, along with a buffer volume, for 2D rendering. The 2D objects drawn on the near plane of the viewing volume comprise the heads-up display (HUD). Everything outside of the near plane and its buffer volume will contain 3D primitives.

Two Unicon 2D graphics functions are of particular interest with regards to the integrated graphics mode: **CopyArea()** and **EraseArea()**. In the 2D facilities, **CopyArea()** copies a specified rectangular area of pixels from a window and pastes it to a destination area, which can be the same or different window. For the integrated facilities, this is a problem because visible 3D primitives in the background could become part of the copied rectangular area. Ideally, in 2D mode **CopyArea()** would only copy 2D, or HUD, elements while ignoring 3D elements. A similar issue exists for **EraseArea()** in the integrated 2D/3D mode. In the 2D facilities, **EraseArea()** fills the specified rectangular area with the background color. For the integrated 2D/3D, there is a danger that using **EraseArea()** for 2D elements could erase 3D elements as an unwanted side-effect. Instead, in integrated 2D/3D mode **EraseArea()** should erase only 2D elements, revealing the 3D elements that may have been obscured behind the 2D elements. See Section 4.2 for more details.

Chapter 4: Implementation

This chapter details and describes the OpenGL implementation of the 2D and integrated 2D/3D facilities. This port is not thread-safe. It was not in the scope of this project to extend Unicon's graphics facilities to support multithreading. Added multithreading support to the OpenGL implementation is a possible avenue for future work.

The bulk of the runtime code for the OpenGL port is in **src/runtime/ropengl2d.ri** and **src/h/opengl.h**. Modifications were made to other runtime source and header files, with most made to **src/runtime/rxwin.ri**, **src/runtime/ropengl.ri**, and **src/h/graphics.h**. Modifications to **src/runtime/rxwin.ri** were in regards to tailoring windowing functions for the OpenGL implementation while leaving the legacy implementation untouched. Modifications to **src/h/graphics.h** were made to add necessary fields to internal structures. The modifications made to **src/runtime/ropengl.ri** were to refactor select code blocks to modularize the graphics facilities. See [12] for more details about the Unicon graphics implementation.

4.1 2D Facilities

Unicon's 2D facilities have an internal runtime API which is detailed in Chapter 8.0 of [12]. To allow for both the legacy Xlib and OpenGL implementations of the 2D facilities to co-exist in the runtime system, this project prepends "**gl_**" (for functions) or "**GL_**" (for macros) to the 2D API. Additionally, the **is_gl** field was added to the internal canvas to identify whether a window is using the Xlib or OpenGL implementation. Macro constants use the **GL2D_** prefix and macro wrappers for OpenGL function which check return values use the **UGL_** prefix.

If this project is deemed a worthy successor, the OpenGL prefixes may be removed and integrated fully into the runtime system. A subset of the internal 2D API, comprised of windowing functions, has been defined in Appendix A.1 for future Unicon developers

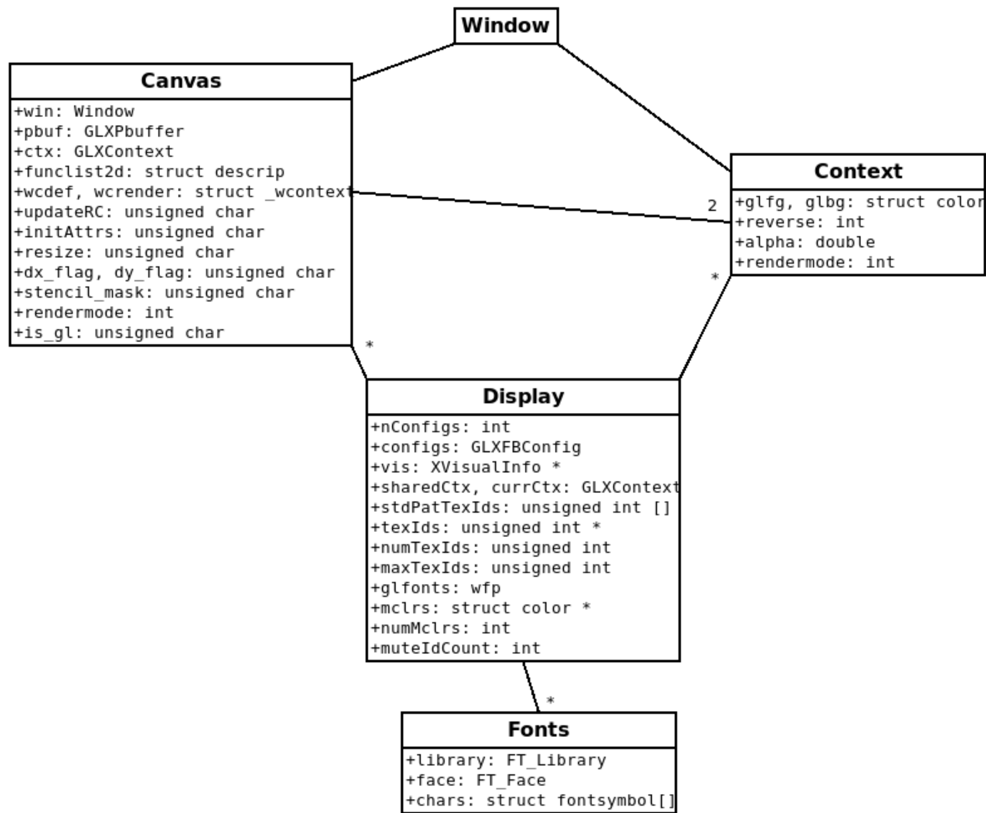

Figure 4.1: UML diagram of the OpenGL implementation of Unicon windows

interfacing OpenGL with other operating systems. Other minor changes to the 2D API may also be found in Appendix A.1.

4.1.1 Windowing

The target operating system for this thesis project is Linux. Conseqently, the X Window System is the target windowing system to be used alongside OpenGL. GLX is an extension to the X Window System which provides an interface between X and OpenGL. The OpenGL rendering context provided by GLX is a **GLXContext**. A **GLXContext** contains all OpenGL state information and can be bound to at most one **GLXDrawable** and one **Display** per thread. A **Display** is the connection provided by the X server to interface with a machine's rendering area.

A **GLXDrawable** is a structure that X can draw into, including a **Window** and a **Pixmap**. **Windows** are onscreen buffer structures and **Pixmaps** are offscreen buffers. Unicon's Xlib implementation uses **Pixmaps** due to X11 lacking a double-buffering mechanism. In keeping with its display list architecture, the OpenGL implementation forgoes using **Pixmaps** and only uses **Windows**. Instead, the OpenGL implementation uses a pixel buffer (**GLXPbuffer**), an OpenGL-specific, offscreen rendering surface, for Unicon hidden windows. As such, the Unicon's internal canvas (**struct _wstate**) owns a **Window**, **GLXPbuffer**, and **GLXContext** to provide an onscreen rendering surface, offscreen rendering surface, and OpenGL context to render with. The decision to give the **GLXContext** to the internal canvas instead of the internal context is due to both the constraints of an integrated 2D/3D window and performance concerns.

A **GLXContext** contains the entire OpenGL state machine, which includes its modelview and projection matrices which determine the landscape and camera position in a 3D scene. In concept, a window that renders a 3D landscape can only have one camera. The only exception is if the parent 3D window creates a child window with its own GLX structures. Furthermore, it would be redundant to allocate multiple OpenGL state machines to render to one window. Unicon makes liberal use of contexts which could lead to memory issues with GLX and OpenGL if asked to allocate too many **GLXContexts**. Memory is cheap, but being liberal with memory not owned by the Unicon runtime system could lead to perplexing undefined behavior.

Unicon's internal display owns a **Display ***, an **XVisualInfo ***, and two **GLXContexts**. One of the GLX contexts is a shared context for sharing textures between all contexts within a display, while the other keeps track of the currently bound context to reduce redundant calls to **glXMakeCurrent()** (see Listing 4.1 for details).

```
#define MakeCurrent(w) do {\
   wsp ws = (w)->window;\
   if (ws->ctx != ws->display->currCtx) {\
      glXMakeCurrent(ws->display->display, ws->win, ws->ctx);\
      ws->display->currCtx = ws->ctx;\
      }\
   } while(0)
```

```
#define UnbindCurrent(wd) do {\
   glXMakeCurrent(wd->display, None, NULL);\
   wd->currCtx = NULL;\
   } while(0)
```

Listing 4.1: Macros for managing and maintaining the current context

Textures are used for all pixel-based rendering. The **XVisualInfo *** contains a visual selected by **glXChooseVisual()** that supports TrueColor or DirectColor, double-buffering, and a depth and stencil buffer. All of the cross-platform display-related OpenGL resources can be found in the internal C function **gl_alc_display()**, with **alc_display()** handles the windowing resources. The internal function that allocates the internal canvas fields and performs window initialization and mapping is **wmap()**. See Listing 4.2 for a list of X and GLX structures used in Unicon internal windowing structures.

```
typedef struct _wstate {
   ...
   GLXContext ctx;
   Window     win;
   ...
} *wsp;

typedef struct _wdisplay {
   ...
   Display   *display;
   XVisualInfo   *vis;
   GLXContext sharedCtx;
   GLXContext currCtx;
   ...
} *wdp;
```

Listing 4.2: GLX fields in internal C structures

A small but significant addition was made to **pollevent()** in **src/runtime/rxwin.ri** and **src/h/graphics.h**. The macro **FLUSH_POLL_INTERVAL** was added to define a counter value for flushing OpenGL instruction every n Unicon virtual machine instructions. An acceptable value was found to be **#define FLUSH_POLL_INTERVAL 10**. It struck a balance between optimizing performance by minimizing the number of calls to **glFlush()** while maintaining good I/O response times. This only applies to Unicon windows that are not buffered, i.e. **"buffer=off"**. This implementation changed **"buffer"** from a context attribute to a canvas attribute. All Unicon windows are not buffered by default.

```
int pollevent()
  {
  ...
  static int gpx_poll = FLUSH_POLL_INTERVAL;
  ...
  if (gpx_poll) gpx_poll--;
  if (!gpx_poll) {
     for (wb = wbndngs; wb; wb=wb->next) {
        wsp ws = wb->window;
        if (ws->buffermode == UGL_IMMEDIATE) {
           if (ws->redraw_flag && !ws->busy_flag) {
              ws->busy_flag = 1;

              MakeCurrent(wb);
              glFlush();

              ws->redraw_flag = 0;
              ws->busy_flag = 0;
              }
           }
        }
     gpx_poll = FLUSH_POLL_INTERVAL;
     }
  ...
  }
```

Listing 4.3: Algorithm for polling flush

The last set of changes affect the implementation of hidden windows for this port. In the Xlib implementation, a hidden window was implemented by moving the target window to some offscreen location. To simplify the process under OpenGL, the window is unmapped using **XUnmapWindow()** and remapped using **XMapWindow()**. It is possible that at some point, unmapping a window did not produce the desired effect. However, using **XUnmapWindow()** and **XMapWindow()** instead of legacy Xlib code resulted in no noticeable change in windowing behavior. Consequently, it appears safe to proceed with a more streamlined approach. These changes were made to internal C function **setcanvas()**.

The OpenGL implementation only supports TrueColor and DirectColor visuals, i.e. colors are represented with RGBA values and the number of colors is not limited (from a practical standpoint). On the other hand, X was designed during a time when PseudoColor and StaticColor visuals (limited to 8-bit colormaps) were the norm. As a consequence, Xlib must allocate colors. This port does not allocate any colors with Xlib except the border and background pixels for the windows managed by the X server. The colors

allocated for the border and background are determined by the initial foreground and background colors, respectively. Upon destruction of a window, the colors are freed. The allocation of Xlib colors occurs in **wmap()**.

4.1.2 Render Context

The Unicon render context is an internal context (**struct _wcontext**) owned by the canvas (**struct _wstate**). The render context maintains the correct context attribute states for the display list in order to render with proper Unicon 2D graphics API semantics, i.e. the render context servers as the Unicon context for a particular Unicon canvas and display list. The context attribute primitives of a display list update the values of the render context when appended to the list and during list traversal. Graphical primitives on the display list are rendered with the values of render context attributes instead of the bound context.

The implementation of updating the render context relies on the internal C function **updaterendercontext()**. **updaterendercontext()** compares each context attribute between the current context and render context. For each attribute that differs, a context attribute item is appended to the display list of the canvas. This function is called from the top of every OpenGL internal C function that allocates a display list record. To prevent redundant recursion from **updaterendercontext()**, the flag **updateRC** was added to the internal canvas. To minimize the number of times **updaterendercontext()** is called, the macro wrapper **UpdateRenderContext()** checks if the serial number of the current context is the same as the context previously used for adding a display list item. The field **lastwcserial** was added to the internal canvas to facilitate this check. **UpdateRenderContext()** has a second argument, the integer code of the context attribute about to be allocated. This prevents an unnecessary display list record allocation if the current attribute value differs from respective render context attribute value.

```
#define UpdateRenderContext(w, intcode) do {\
   if (!(w)->window->initAttrs && !(w)->window->updateRC &&\
       (w)->window->lastwcserial != (w)->context->serial)\
```

```
    {\
    int rv = updaterendercontext(w, intcode);\
    if (rv != Succeeded) return rv;\
    }\
 } while(0)
```

Listing 4.4: Macro wrapping for updating the Unicon render context

UpdateRenderContext() is implemented with the display list design mentioned in Section 3.3. Calls to **updaterendercontext()** scale directly with the number of differing contexts used with one canvas. The speed of **updaterendercontext()** scales directly with the number of differing context attributes between the current context and the Unicon render context. **updaterendercontext()** is never called if exactly one context is used for every canvas. According to profiling via **gprof** on Unicon standard programs **IVIB** and **UI**, both the profiling time and number of calls to **updaterendercontext()** is relatively insignificant. Both **IVIB** and **UI** are written with Unicon's GUI toolkit which heavily use Unicon procedure **Clone()**, and consequently multiple contexts per canvas.

Another consideration of the render context is initialization. Whenever a redraw is invoked, the render context needs to be reinitialized to ensure the correct context attributes apply to graphical primitives on the display list that are not preceded by context attribute items. Another internal context, **wcdef**, is used to remember the initial context attributes for a canvas. The initial, or default, context **wcdef** takes the values of the context allocated and initialized with user-defined arguments when a new window is opened with the internal C function **gl_wopen**. The function **copy_2dcontext()** is used for initialization of both **wcdef** and **wcrender** and reinitialization of **wcrender** during redraws. To initialize context attributes without adding display list records, the flag **initAttrs** was added to the internal canvas. All of the fields added to the internal canvas for the render context can be seen below in Listing 4.5.

```
typedef struct _wstate {
   ...
   struct _wcontext wcrender, wcdef;
   int lastwcserial, updateRC, initAttrs;
   ...
} *wsp;
```

Listing 4.5: Internal canvas render context fields

4.1.3 Display List

The runtime architecture of the new 2D implementation is modeled after the 3D facilities. Unicon's 2D display list contains 18 graphical primitives and 14 context attributes primitives. However, one major difference is that each display list object is a Unicon record, whereas the display list objects of the 3D facilities are either Unicon records or lists. The choice to define all 2D display list objects as Unicon records was in equal parts to reduce the number of runtime typechecks during display list traversal in additon to simplifying display list inspection and modification by the end user. See Appendix A.2 for a detailed list of the 2D display list objects and which Unicon procedures they are instantiated from.

Unicon procedures which accept multiples of arguments— **DrawArc()**, **DrawCircle()**, **DrawRectangle()**, **DrawString()**, **EraseArea()**, **FillArc()**, **FillCircle()**, **FillRectangle()**, **WWrite()**, and **WWrites()**— create distinct display list objects for each instance specified in the argument list. A possible avenue for future work is to determine if and at what point it is beneficial to performance to bundle multiple display list objects of the same type into one display list object.

The internal C functions **create_display_list2d()**, **rec_structor2d()**, and **traversefunclist2d()** are 2D analogs to the 3D internal functions that create a Unicon list for the display list, return Unicon record constructors for allocating display list records, and render the items on the display list. The macro **Get2dRecordConstr()** wraps around **rec_structor2d()** for convenience and is used in each internal function of the 2D API that pertains to a particular graphical primitive or context attribute, e.g. **gl_fillpolygon()** and **gl_setfg()**. Whenever one of these internal C functions is invoked, a Unicon record of the corresponding type is allocated, initialized and appended to the display list. The display list traversal algorithm for the 2D mode is a simple list traversal that invokes the appropriate functions to render each display list record.

On the other hand, the traversal algorithm for the integrated 2D/3D mode is much more interesting because of **EraseArea()** semantics. The field **stencil_mask** was added to the internal canvas to allow the integrated 2D/3D mode traversal to implement the correct **EraseArea()** semantics.

The display list is traversed while only invoking functions of context attribute records to keep the Unicon render context up-to-date. If no **EraseArea** primitive is encountered, the previously saved render context is restored and the display list is traversed normally. If an **EraseArea** primitive is encountered, then its function is invoked, which sets all areas of the erase bitplane except for the specified rectangular area to be erased. The internal canvas stencil mask **stencil_mask** is set to the erase bitplane, the appropriate stencil function is set, and the previously saved render context is used. Normal display list traversal then occurs starting from the last saved place in the list (either the start, or right after the last **EraseArea** primitive). After this partial traversal is done, the stencil mask is reset to 0 and the current render context and position in the display list is saved. This repeats until the entire list is traversed.

The internal function **traverselist2d()** is a helper function used by **traversefunclist2d()** that traverses a specified contiguous subset of the 2D display list. Listing 4.6 depicts pseudo-code for 2D display list traversal used by **traversefunclist2d()**.

```
int traversefunclist2d(wbp w)
   {
   wsp ws = w->window;
   wcp wc = w->context;
   int i, j, intcode;
   int used;
   word k;
   int elements;
   struct b_list *hp;
   tended struct b_lelem *bp;
   int rv;
   ...
   if (ws->is_3D) { /* integrated 2d/3d mode traversal */
      /*
       * Initialize render context and traversal state
       */
      struct descrip desc;
      struct b_record *rp;
      wcp wcr = &(ws->wcrender);
      word last_k = k;
      int last_used = used, last_i;
```

```
   tended struct b_lelem *last_bp = bp;
   struct _wcontext last_wc = {0};
   copy_2dcontext(&last_wc, wcr);

   for (i=0, last_i=0; i< elements; i++) {
      ...
      if (intcode == GL2D_ERASEAREA) {
         /*
          * Draw area to be erased into stencil buffer
          */
         erasearea2d(w, rp);

         /*
          * Setup stencil buffer and render context to traverse all
          * contiguous non-erasearea display list records before this
          */
         ws->stencil_mask = GL2D_ERASE_BIT;
         DefaultStencilFunc(w);
         copy_2dcontext(wcr, &last_wc);
         init_2dcanvas(w);
         rv = traverselist2d(w, last_bp, last_i, i, last_used, last_k);
         if (rv != Succeeded) return rv;

         /*
          * Set stencil buffer states back to default and save the current
          * render context state and position in display list
          */
         ws->stencil_mask = 0;
         DefaultStencilFunc(w);
         copy_2dcontext(&last_wc, wcr);
         last_k = k;
         last_i = i+1;
         last_used = used;
         last_bp = bp;
         }
      else if (i == elements-1 && last_i < i) {
         /*
          * Traverse the remaining non-erasearea elements of the list
          */
         copy_2dcontext(wcr, &last_wc);
         rv = traverselist2d(w, last_bp, last_i, i+1, last_used, last_k);
         if (rv != Succeeded) return rv;
         }
      else {
         /*
          * Only traverse context attributes to update the render context
          */
         ...
         } /* end else */
      } /* end for */
   } /* end if */
else { /* 2d traversal */
   /*
    * Traverse all elements normally
    */
   rv = traverselist2d(w, bp, 0, elements, used, k);
   if (rv != Succeeded) return rv;
   }

if (ws->resize) ws->resize = 0;
if (ws->redraw_flag) ws->redraw_flag = 0;
return Succeeded;
}
```

Listing 4.6: Code for display list traversal

A few flags (**resize**, **dx_flag**, **dy_flag**) were added to the internal canvas to facilitate correct window resizing semantics in addition to enabling users to modify particular fields of display list records. See Appendix B.3 for a list of display list records and their modifiable fields. The macros **Recalc*()** and **Update*()** determine whether the private fields of a display list record need to be updated and update the private fields respectively.

4.1.4 Textures

The implementation of Unicon's 2D pixel-based writing operations (**CopyArea()**, **DrawImage()**, **ReadImage()**, **DrawString()** and **WWrite*()**) uses OpenGL textures. There is the option of using OpenGL function **glDrawPixels()**, but it is significantly slower than loading an image to a 2D texture and rendering it to a quad. In fact, even the OpenGL function **glReadPixels()** is a bottleneck for performance because it forces CPU and GPU sychronization. A possible avenue for future work is to implement OpenGL pixel buffer objects (available starting OpenGL version 2.1) which allow for asynchronous pixel transfer.

As mentioned briefly in Section 4.1.1, the internal Unicon display owns a **GLXContext** for the sole purpose of sharing textures to all OpenGL rendering contexts for a particular display. Additional fields were added to the internal display structure for storing the textures allocated from OpenGL.

```
typedef struct _wdisplay {
   ...
   unsigned int  stdPatTexIds[16];      /* array of std pattern texture ids */
   unsigned int  *texIds;
   unsigned int  numTexIds;
   unsigned int  maxTexIds;
   ...
} wdp;
```

Listing 4.7: Fields in internal display structure for storing OpenGL textures

OpenGL texture IDs are integer values ranging from 0 to 65535, with 0 being reserved as the default texture for all OpenGL texture targets, i.e. **glGenTextures()** will never return a texture ID of 0. **stdPatTexIds** holds the texture IDs that have been initialized to the standard Unicon patterns. **texIds** holds all other textures that are allocated by

the 2D facilities. **numTexIds** stores the number of allocated texture IDs held by **texIds** and **maxTexIds** indicates size of **texIds**.

stdPatTexIds and **texIds** are arrays maintained in such a way that a value of 0 indicates that no texture has been allocated for that slot. Internal C function **get_tex_index()** dynamically manages the size of **texIds** and returns the first available index of the array that is ready to be allocated an OpenGL texture ID. The macro **InitTexture2d()** handles the allocation of the texture into the provided index of **texIds**.

```
/*
 * Allocates {ntex} OpenGL textures. If more than one are requested, the
 * index returned is the first of the requested array.
 */
int get_tex_index(wdp wd, unsigned int ntex)
   {
   int rv;
   unsigned int err;
   /* Max # is dependent on GLuint (65536-1), but reserve a fraction */
   const unsigned int MAX_TEXTURES = 32768;

   if (ntex == 0) return Failed;

   if (wd->numTexIds+ntex > wd->maxTexIds) {
      unsigned int *ptr = NULL;
      unsigned int n = wd->maxTexIds;

      if (wd->maxTexIds < MAX_TEXTURES) {
         ptr = (unsigned int *)realloc((wd)->texIds,2*n*sizeof(unsigned int));
         }

      if (!ptr) {
         /* if no memory left, delete half of the existing textures */
         delete_first_tex(wd, n/2);
         }
      else {
         wd->texIds= ptr;
         wd->maxTexIds = 2*n;
         }
      }
   rv = wd->numTexIds;
   wd->numTexIds += ntex;
   return rv;
   }


#define InitTexture2d(texptr) do {\
   glGenTextures(1, texptr);\
   UGLBindTexture(GL_TEXTURE_2D, *texptr);\
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);\
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);\
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);\
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);\
   glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);\
   } while(0)
```

Listing 4.8: Internal procedures for texture allocation

If the maximum number of textures is reached, **get_tex_index()** releases the first half of the allocated texture IDs using internal function **delete_first_tex()**. **delete_first_tex()** releases textures by using **glDeleteTextures()**, moving the remaining textures to the front of the array and setting the remaining slots of **texIds** to 0. **numTexIds** is adjusted accordingly. **texIds** is always filled from front to back to facilitate ease of texture deallocation.

```
/*
 * Frees the first {ntex} (OpenGL) resources in the
 * texture array moves the remaining (if any) to the front
 * of the array.
 */
int delete_first_tex(wdp wd, unsigned int ndel)
   {
   unsigned int i, j, ntex = wd->numTexIds;
   unsigned int *a;

   if (ndel == 0) return Failed;
   else if (ndel > ntex) ndel = ntex;

   a = wd->texIds;
   j = ntex - ndel;

   glDeleteTextures(ndel, a);
   wd->numTexIds = j;

   for (i = 0; i < ndel; i++) {
      if (j < ntex) {
         a[i] = a[j];
         a[j++] = 0;
         }
      else
         a[i] = 0;
      }
   return Succeeded;
   }
```

Listing 4.9: Internal function for texture deallocation

4.1.5 Stencil Buffer

The stencil buffer is used in several aspects of this OpenGL implementation of the Unicon 2D graphics facilities. The stencil buffer, like the color and depth buffers, is a memory buffer the size of the screen (for OpenGL, defined by **glViewport()**). The stencil buffer can be at most 8 bits deep. Each bit of depth is referred to as a bitplane. The visual given by **glXChooseVisual()** determines the actual stencil buffer depth. The Unicon OpenGL implementation needs only 3 bitplanes, one each for general-purpose drawing,

patterns, and erasearea (for integrated 2D/3D mode).

```
#define GL2D_DRAW_BIT 0x01
#define GL2D_PATT_BIT 0x02
#define GL2D_ERASE_BIT 0x04
```

Listing 4.10: The 3 bitplanes defined for Unicon's 2D graphics facilities

The purpose of the stencil buffer is to cull particular pixels of a rendering operation. The culling is affected by both the values present in the stencil buffer and the comparison function used. The utility of the stencil buffer is vast and not fully explored in this implementation. To facilitate ease of use in the runtime system, four macros were written as wrappers for OpenGL functions to utilize the stencil buffer: **EnableStencilWrite()**, **DisableStencilWrite()**, **SetStencilFunc()**, and **DefaultStencilFunc()**.

EnableStencilWrite() enables writing to a specified bitplane or bitplanes and clears the specified bitplane(s) to a given value. Lastly, **EnableStencilWrite()** defines what type of write action to be to applied to affect portions of the stencil buffer when rendering operations occur. **DisableStencilWrite()** in contrast disables any rendering operations from affecting values in the stencil buffer.

```
/*
 * {mask} determines the bitplane(s) affected.
 * {val} is the value the stencil buffer is cleared to.
 * {action} determines the the action to be taken when the stencil
 * test passes.
 */
#define EnableStencilWrite(mask, val, action) do {\
   glStencilMask(mask);\
   glStencilOp(GL_KEEP, GL_KEEP, action);\
   glClearStencil(val);\
   glClear(GL_STENCIL_BUFFER_BIT);\
   } while(0)

#define DisableStencilWrite() do {\
   glStencilMask(0x0);\
   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);\
   } while(0)
```

Listing 4.11: Macros for controlling writes to the stencil buffer

SetStencilFunc() enables a subset of the 3 available bitplanes to affect rendering operations. **DefaultStencilFunc()** sets the stencil function to the default stencil mask, **stencil_mask**, a newly added field to the internal canvas. By default, **stencil_mask = 0**, meaning that although the stencil test is enabled, the values in the stencil buffer will

not affect rendering operations. The only time when **stencil_mask** holds a different value is during display list traversal of an integrated 2D/3D mode window when EraseArea() is called with arguments not affecting the entire screen. In that case, **stencil_mask** is set to **GL2D_ERASE_BIT**. See Section 4.1.3 for more details.

```
typedef struct _wstate {
   ...
   unsigned char stencil_mask;
   ...
} *wsp;

/*
 * glStencilFunc() with a mask of 0 lets the stencil test pass regardless
 * of what values are in the stencil buffer. Otherwise, it will only let
 * pixels that are equal to the bits in {mask} pass the stencil test.
 */
#define SetStencilFunc(w, mask) \
   glStencilFunc(GL_EQUAL, 0xFF, (w)->window->stencil_mask|mask)
#define DefaultStencilFunc(w) SetStencilFunc(w, 0)
```

Listing 4.12: Macros for applying the stencil buffer

4.1.6 Context Attributes

Context attributes affect the rendering of graphical primitives. There are five categories of attributes: color, font, drawing style, clipping, and translation. This section will describe how the Unicon attribute semantics were achieved with OpenGL.

Colors

As discussed at the end of Section 4.1.3, this OpenGL implementation is limited to support of 24-bit TrueColor and DirectColor displays. OpenGL's RGBA mode offers a simple way to change color states with the **glColor*()** class of functions, which take RGB or RGBA values as a variety of C data types.

```
typedef struct color {
   char name[6+MAXCOLORNAME];
   unsigned short r, g, b, a;
   int id;
#ifdef XWindows
   unsigned long c; /* X11 color handle */
#endif                                /* XWindows */
   struct color *prev, *next;
   } *clrp;
```

Listing 4.13: OpenGL color struct

struct color (see Listing 4.13) is similar to the internal structure used by the Xlib implementation, **struct wcolor**. The difference between the structures is that **struct wcolor** uses closed hashing while **struct color** uses external hashing. If **id** is 0, then the internal RGBA values are used. Otherwise, it holds a positive integer that references an item in the mutable color linked list within the internal display structure as seen in Listing 4.14. The convenience macros **SetColor()**, **GetColor*()**, and **CopyColor()** were written to interface with **struct color**. The **c** field exists to interface with legacy Xlib color code for the border and background pixel values of windows provided by X11.

```
typedef struct _wdisplay {
   ...
   int           numMclrs;
   int           muteIdCount;
   struct color *mclrs;
   ...
} *wdp;
```

Listing 4.14: OpenGL mutable color fields

These new structures are integral for the two new fields in the internal context, **glfg** and **glbg** (see Listing 4.15), that hold the current foreground and background colors respectively. If either **glfg** or **glbg** is a mutable color, then the RGBA values of the corresponding item in the mutable color linked list is used. Otherwise, the RGBA values in **glfg** or **glbg** are used. The function **gl_color()** handles allocation of the foreground and background display list objects. **setcolor2d()** handles updating the internal context foreground and background states.

```
typedef struct _wcontext {
   ...
   struct color glfg, glbg;
   ...
} *wcp;
```

Listing 4.15: New foreground/background context fields

All five color context attributes (**fg**, **bg**, **reverse**, **gamma** and **drawop**), are interlinked with how they affect the OpenGL color state. The macro **SetDrawopColorState()** accounts for each of these attributes and is called after each to update the OpenGL color state.

```
#define SetDrawopColorState(w, drawop, is_fg) do {\
   wsp ws = (w)->window;\
   wcp wc = (w)->context, wcr = &(ws->wcrender);\
   int is_reverse;\
   unsigned short color[4], fg[4], bg[4];\
\
   if (is_fg)\
      GetContextColorUS(w, FG, color[0], color[1], color[2], color[3]);\
   else\
      GetContextColorUS(w, BG, color[0], color[1], color[2], color[3]);\
\
   /* set fg to (fg ^ bg) */\
   if (drawop == GL2D_DRAWOP_REVERSE) {\
      if (is_fg)\
         GetContextColorUS(w, BG, bg[0], bg[1], bg[2], bg[3]);\
      else\
         GetContextColorUS(w, FG, bg[0], bg[1], bg[2], bg[3]);\
\
      /* calculate xor */\
      color[0] = color[0] ^ bg[0];\
      color[1] = color[1] ^ bg[1];\
      color[2] = color[2] ^ bg[2];\
      color[3] = 65535; /* transparency is disabled for "xor" */\
      }\
   SetColorState(color, wcr->gamma);\
   } while(0)
```

Listing 4.16: The macro responsible for setting the OpenGL color state

Alpha **alpha** is a new context attribute for the Unicon's 2D facilities that implements transparency. For the OpenGL implementation, it is not a render context attribute in the fact that no display list record is created for this attribute. Instead, it is a normal Unicon context attribute that is used as a default alpha value for any Unicon color specifications missing an alpha. If an alpha value is explicitly mentioned in a Unicon color specification, then **alpha** is ignored.

Reverse and Gamma The **reverse** attribute has been added as a separate field to the internal context, **reverse**, to support the render context. The implementation relies on a few macros that retrieve the color (foreground or background) or set the OpenGL color state from the current context depending on the state of **reverse**.

A set of macros were written to support the encoding and decoding of **gamma** to convert to and from linear and sRGB space. These macros follow the form of **EncodeGamma*()** and **DecodeGamma*()**. There are variants specifying accepted C

data types—unsigned short (US), unsigned char (UC), and float (GLF)—in addition to the size of array parameters. **gamma** shares top-level macros with **reverse** to provide correct Unicon color semantics.

```
#define EncodeGamma(flt, gamma) pow(flt, 1/gamma)
#define DecodeGamma(flt, gamma) pow(flt, gamma)

#define EncodeGammaUSToUC(ushort, gamma) \
   (unsigned char)(GLFToUC(EncodeGamma(USToGLF(ushort),gamma)))
#define DecodeGammaUSToUC(ushort, gamma) \
   (unsigned char)(GLFToUC(DecodeGamma(USToGLF(ushort),gamma)))

#define EncodeGammaUS(ushort, gamma) \
   (unsigned short)(GLFToUS(EncodeGamma(USToGLF(ushort),gamma)))
#define EncodeGammaUS_V4(dest, src, gamma) do {\
   dest[0] = EncodeGammaUS(src[0], gamma);\
   dest[1] = EncodeGammaUS(src[1], gamma);\
   dest[2] = EncodeGammaUS(src[2], gamma);\
   dest[3] = src[3];\
   } while(0)

#define EncodeGammaUSToUC_V4(dest, src, gamma) do {\
   dest[0] = EncodeGammaUS(src[0], gamma) >> 8;\
   dest[1] = EncodeGammaUS(src[1], gamma) >> 8;\
   dest[2] = EncodeGammaUS(src[2], gamma) >> 8;\
   dest[3] = src[3] >> 8;\
   } while(0)

#define DecodeGammaUS(ushort, gamma) \
   (unsigned short) GLFToUS(DecodeGamma(USToGLF(ushort), gamma))

#define DecodeGammaUC(uchar, gamma) \
   (unsigned char) GLFToUC(DecodeGamma(UCToGLF(uchar), gamma))

#define EncodeGamma_V4(dest, src, gamma) do {\
   dest[0] = EncodeGamma(src[0], gamma);\
   dest[1] = EncodeGamma(src[1], gamma);\
   dest[2] = EncodeGamma(src[2], gamma);\
   dest[3] = src[3];\
   } while(0)

#define DecodeGamma_V4(dest, src, gamma) do {\
   dest[0] = DecodeGamma(src[0], gamma);\
   dest[1] = DecodeGamma(src[1], gamma);\
   dest[2] = DecodeGamma(src[2], gamma);\
   dest[3] = src[3];\
   } while(0)
```

Listing 4.17: Macros for encoding and decoding **gamma**

SetDrawopColorState() sets the OpenGL color state specifying the foreground or background color and drawop to be applied. The **GetContextColor*()** family of macros retrieves the values of either the foreground or background color as the specified C data type while taking into account the reverse attribute.

```
#define GetContextColorUS(w, is_fg, r, g, b, a) do {\
   wcp wcr = &((w)->window->wcrender);\
   if (wcr->reverse) {\
      if (is_fg)\
         GetColorUS(w, wcr->glbg, r, g, b, a);\
      else\
         GetColorUS(w, wcr->glfg, r, g, b, a);\
      }\
   else {\
      if (is_fg)\
         GetColorUS(w, wcr->glfg, r, g, b, a);\
      else\
         GetColorUS(w, wcr->glbg, r, g, b, a);\
      }\
   } while(0)

#define GetContextColorUC(w, is_fg, r, g, b, a) do {\
   wcp wcr = &((w)->window->wcrender);\
   if (wcr->reverse) {\
      if (is_fg)\
         GetColorUC(w, wcr->glbg, r, g, b, a);\
      else\
         GetColorUC(w, wcr->glfg, r, g, b, a);\
      }\
   else {\
      if (is_fg)\
         GetColorUC(w, wcr->glfg, r, g, b, a);\
      else\
         GetColorUC(w, wcr->glbg, r, g, b, a);\
      }\
   } while(0)
```

Listing 4.18: Get macros for struct color

Finally, **SetColorState()** is the macro responsible for encoding the **gamma** and setting the OpenGL color state. **GetContextColor*()** retrieves the values of either the foreground or background color as the specified C data type while taking into account the **reverse** attribute. It should be mentioned that **reverse** does not modify the semantics of setting the unicon attributes **fg** or **bg** via **WAttrib()**, **Fg()**, or **Bg()**. It only affects which colors are used for rendering geometric primitives, bitmaps, and pixmaps.

```
#define SetColorState(color, gamma) do {\
   GLfloat norm[4], normg[4];\
   USToGLF_V4(norm, color);\
   EncodeGamma_V4(normg, norm, gamma);\
   glColor4fv(normg);\
   } while(0)
```

Listing 4.19: The macro for encoding **gamma** and setting the OpenGL color state

Drawop This implementation of the 2D facilities only supports the values **"copy"** and **"reverse"** for **drawop**. OpenGL provides the function **glLogicOp()** to allow for

the selection of a specific logical operation to be applied to the incoming and buffer color values. To enable OpenGL logic operations, **GL_COLOR_LOGIC_OP** must be enabled. A consequence of enabling **GL_COLOR_LOGIC_OP** is that blending is disabled, which is needed for alpha blending, or transparency.

"drawop=copy" is implemented by disabling **GL_COLOR_LOGIC_OP** and allowing normal pixel application. As long as no transparency is used (**"alpha=1.0"**), pixels will have a one-to-one application to the color buffer. **"drawop=reverse"** is implemented by initializing the OpenGL logic operation to XOR by using **glLogicOp()** and enabling **GL_COLOR_LOGIC_OP**. **SetDrawopColorState()** will then XOR the corresponding foreground and background components together (excluding alpha) and set the OpenGL color state with the resulting color using **SetColorState()**. See 4.16 for details. The macro **ApplyDrawop()** is used to apply these state changes.

```
#define ApplyDrawop(w, drawop) do {\
   if (drawop == GL2D_DRAWOP_COPY) {\
      glDisable(GL_COLOR_LOGIC_OP);\
      }\
   else if (drawop == GL2D_DRAWOP_REVERSE) {\
      glEnable(GL_COLOR_LOGIC_OP);\
      }\
   SetDrawopColorState(w, drawop, FG);\
   } while(0)
```

Listing 4.20: The macro for applying **drawop** to the OpenGL color state

Font

The font rendering facilities were implemented by using FreeType, an open-source font rasterization library, to provide glyph images and OpenGL to render those images. The internal C function **drawstringhelper()** handles the retrieval, storage, and rendering of glyph images while the internal C functions **gl_loadfont()** and **find_fontfile()** find the best fitting font file from Unicon's standard fonts in **unicon/dat/fonts** and allocate an internal font structure for the context to reference. Four open-source font files were added to **unicon/dat/fonts** to fill the default **"mono"** font: **SourceCodePro-Regular.ttf**, **SourceCodePro-It.ttf**, **SourceCodePro-Bold.ttf**, and **SourceCodePro-BoldIt.ttf**.

The font dimension attributes for a selected font—width, height, ascent, and descent—are fixed. However, the user may change the **leading** attribute, which specifies the amount of vertical space the cursor travels when encountering a newline. No change was made to how the leading attribute was implemented, except for changing font attribute retrieval macros to reflect FreeType's data structures.

```
#define FT_ASCENT(face) (int)((face)->size->metrics.ascender >> 6)
#define FT_DESCENT(face) (int)((face)->size->metrics.descender >> 6)
#define FT_FWIDTH(face) (int)((face)->size->metrics.max_advance >> 6)
#define FT_FHEIGHT(face) \
   (int)(((face)->size->metrics.ascender-(face)->size->metrics.descender) >> 6)
#define GL_ASCENT(w) FT_ASCENT((w)->context->font->face)
#define GL_DESCENT(w) FT_DESCENT((w)->context->font->face)
#define GL_FWIDTH(w) FT_FWIDTH((w)->context->font->face)
#define GL_FHEIGHT(w) FT_FHEIGHT((w)->context->font->face)
#define GL_LEADING(w) ((w)->context->leading)
#define GL_ROWTOY(w,row) ((row-1) * GL_LEADING(w) + GL_ASCENT(w))
#define GL_COLTOX(w,col) ((col-1) * GL_FWIDTH(w))
#define GL_YTOROW(w,y) ((y>0) ? ((y)/GL_LEADING(w)+1) : ((y)/GL_LEADING(w)))
#define GL_XTOCOL(w,x) (!GL_FWIDTH(w) ? (x) : ((x) / GL_FWIDTH(w)))
#define GL_MAXDESCENDER(w) GL_DESCENT(w)
#define GL_TEXTWIDTH(w, s, len) drawstringhelper(w, 0, 0, 0, s, len, 0, 0)
```

Listing 4.21: OpenGL implementation of Unicon font macros

Drawing Style

Linewidth OpenGL provides the function **glLineWidth()** which is only guaranteed to support a width of 1. The implementation of OpenGL determines the range of widths supported. At the moment, **glLineWidth()** is used to implement line width. The macro **ApplyLinewidth()** is a wrapper for **glLineWidth()**. This is another avenue for future work if the limited linewidth becomes a problem.

Linestyle The **linestyle** attribute is implemented as the rendering macro **RenderRealarrayLinestyle()** used by the internal rendering function **drawgeometry2d()**. Each time a graphical line primitive is allocated, another set of vertices containing a segmented, or dashed, version of the original line is created. The segmented vertices of circles and arcs are created by selectively sampling the original vertices.

```
int drawgeometry2d(wbp w, struct b_record *rp)
   {
```

```
    ...
    /* calculate points */
    for (j = i, k = 1; j < n; j+=2) {
       v[j] = wx + rx*cos(start);
       v[j+1] = wy + ry*sin(-start); /* to imitate X11 rotation */

       if (!fill && ((j-i)/2 \% SKIP == 0) && k < nseg) {
          vseg[k] = v[j];
          vseg[k+1] = v[j+1];
          k+=2;
          }
       start += dtheta;
       }

    /* calculate endpoint specifically */
    v[n-2] = wx + rx*cos(end);
    v[n-1] = wy + ry*sin(-end); /* to imitate X11 rotation */
    if (!fill) {
       vseg[nseg-2] = v[n-2];
       vseg[nseg-1] = v[n-1];
       }
    break;
    }
    ...
}
```

Listing 4.22: Selective vertex sampling to segment circles and arcs

All other line primitves (rectangles, lines, line segments, and polygons) are segmented with the function **segment_line()**. For each line segment given in a set of lines, **segment_line()** calculates the size of the segmented array based on a fixed line segment size. The new segmented array is allocated and filled with newly calculated vertices.

```
struct b_list *segment_line(wbp w, int num, struct b_realarray *ap2,
   double *v2, word n2)
   {
   wsp ws = (w)->window;
   wcp wcr = &(ws->wcrender);
   tended struct b_realarray *apseg, *ap2_t = ap2;
   tended struct descrip desc;
   word i, j, k, nseg;
   double dx, dy, len, *vseg;
   const double MIN_LEN = 4.0;

   if (n2 \% 2) {
      return NULL;
      }

   if (ap2_t) v2 = ap2_t->a;

   dx = wcr->dx;
   dy = wcr->dy;
   nseg = 0;

   /* Find the size of the segmented array */
   for (i = 0; i < n2; i+=num) {
      if (i+3 < n2) {
         double deltax = v2[i+2]-v2[i];
         double deltay = v2[i+3]-v2[i+1];
         len = sqrt(deltax*deltax + deltay*deltay);
```

```
      nseg += (word)(2*ceil(len/MIN_LEN)); /* x2 for 2d coords */
      }
   else if (i+1 < n2) {
      /* last vertex, add it to the segmented array */
      nseg += 2;
      }
   }

/* Alloc segmented array */
nseg++; /* +1 for drawcode */
AlcRealarrayList(desc, apseg, nseg, NULL);
vseg = apseg->a;
vseg[0] = GL_LINES;
if (ap2_t) v2 = ap2_t->a;

/* Calculate points and place them on real array */
for (i = 0, j = 1; i < n2; i+=num) {
   if (i+3 < n2) {
      /* two vertices available, segment the line */
      word n;
      double deltax, deltay, theta;
      deltax = v2[i+2]-v2[i];
      deltay = v2[i+3]-v2[i+1];
      theta = atan2(deltay, deltax);

      /* maybe use macros to make this faster */
      len = sqrt(deltax*deltax + deltay*deltay);
      n = (word)(2*ceil(len/MIN_LEN));

      /* add segmented vertices to array */
      for (k = 0; k < n; k+=2) {
         double delx = (k/2)*MIN_LEN*cos(theta);
         double dely = (k/2)*MIN_LEN*sin(theta);
         vseg[j+k] = GLWORLDCOORD_RENDER_X(w, v2[i]+dx+delx);
         vseg[j+k+1] = GLWORLDCOORD_RENDER_Y(w, v2[i+1]+dy+dely);
         }
      j+=n;
      }
   else if (i+1 < n2) {
      /* add last vertex to array */
      vseg[j] = GLWORLDCOORD_RENDER_X(w, v2[i]+dx);
      vseg[j+1] = GLWORLDCOORD_RENDER_Y(w, v2[i+1]+dy);
      j+=2;
      }
   }
return (struct b_list *)BlkLoc(desc);
}
```

Listing 4.23: The internal function for segmenting straight line segments

At render time, a runtime check for the **linestyle** attribute determines which set of vertices are rendered and how. If **"linestyle=solid"**, then the original set of vertices is rendered. If **"linestyle=dashed"**, then the segmented set of vertices is rendered. If **"linestyle=striped"**, then the origin set of vertices is rendered with the current background color followed by the segmented set of vertices with the foreground color.

```
#define RenderRealarrayLinestyle(w, v, n, vseg, nseg) do {\
   int linestyle = (w)->window->wcrender.linestyle;\
```

```
  if (linestyle == GL2D_LINE_SOLID) {\
     RenderRealarray(v, n);\
     }\
  else {\
     if (linestyle == GL2D_LINE_STRIPED) {\
        int drawop = (w)->window->wcrender.drawop;\
        SetDrawopColorState(w, drawop, BG);\
        RenderRealarray(v, n); \
        SetDrawopColorState(w, drawop, FG);\
        }\
     RenderRealarray(vseg, nseg);\
     }\
  } while(0)
  }
```

Listing 4.24: Implementation of attribute **linestyle**

Fillstyle and Pattern The **fillstyle** is implemented by the rendering macro **RenderRealarrayFillstyle()** used by the internal C function **drawgeometry2d()**, like the **linestyle**. A render time check on the Unicon render context's value of the **fillstyle** is used to determine how to render the primitive. The pattern bitplane of the stencil buffer is used to render filled primitives with **"fillstyle=masked"** and **"fillstyle=textured"**. The only difference being that **"fillstyle=textured"** renders the filled primitive (without the stencil patter bitplane) using the background color before rendering the primitive using the foreground color and stencil pattern bitplane.

```
#define RenderRealarrayFillstyle(w, v, n, stencil_mask) do {\
  switch ((w)->window->wcrender.fillstyle) {\
     case GL2D_FILL_SOLID:\
        SetStencilFunc(w, stencil_mask);\
        RenderRealarray(v, n);\
        break;\
     case GL2D_FILL_MASKED:\
        SetStencilFunc(w, stencil_mask | GL2D_PATT_BIT);\
        RenderRealarray(v, n);\
        DefaultStencilFunc(w);\
        break;\
     case GL2D_FILL_TEXTURED:\
        SetStencilFunc(w, stencil_mask);\
        SetDrawopColorState(w, wcr->drawop, BG);\
        RenderRealarray(v, n);\
        SetStencilFunc(w, stencil_mask | GL2D_PATT_BIT);\
        SetDrawopColorState(w, wcr->drawop, FG);\
        RenderRealarray(v, n);\
        break;\
     default:\
        return Failed;\
        break;\
     }\
  DefaultStencilFunc(w);\
  } while(0)
```

Listing 4.25: Implementation of attribute **fillstyle**

As can be seen above from Listing 4.25, when When **“fillstyle=masked”** or **“fillstyle=textured”**, the stencil function is changed to apply the pattern bitplane, which allows pixels drawn to the shape of the pattern bitplane to pass and be rendered to the color buffer.

The pattern attribute is implemented in internal C function **setpattern2d()** by using textures to draw the pattern stencil buffer bitplane. To be more specific, a bitmap is created from either a standard Unicon pattern or a specification string and converted to an alpha-based pixmap by **bitmap_to_pixmap()**. A texture is allocated and initialized with the alpha-based pixmap. Alpha testing and OpenGL texture repeating are used to draw the pattern into the entire pattern bitplane of the stencil buffer. The color and depth masks are disabled to prevent the pattern from affecting the color or depth buffers. **SetStencilFunc()** can then be used to specify that the pattern bitplane should be applied to a render operation.

```
int setpattern2d(wbp w, struct b_record *rp)
   {
   wsp ws = w->window;
   wcp wcr = &(ws->wcrender);
   wdp wd = ws->display;
   int width, nbits, size, slen, s1len, update;
   unsigned int index = 0;
   char *pattern;
   char *s, *bits, *s1;
   unsigned int texid = 0, *texptr = NULL;

   if (rp) {
      GetStr(rp->fields[2], s, slen);
      GetStr(rp->fields[3], s1, s1len);

      /* Check if pattern string has changed */
      update = (slen <= s1len) ? strncmp(s, s1, slen) : strncmp(s, s1, s1len);
      if (update) {
         MakeStr(s, slen, &(rp->fields[3]));
         }
      else {
         if (is:integer(rp->fields[6]) && is:integer(rp->fields[7])) {
            texid = (unsigned int) IntVal(rp->fields[6]);
            index = (unsigned int) IntVal(rp->fields[7]);

            /* Check if non-std pattern needs to be reallocated */
            if (index && texid != wd->texIds[index]) {
               update = 1;
               }
            else {
               width = IntVal(rp->fields[4]);
               nbits = IntVal(rp->fields[5]);
               UGLBindTexture(GL_TEXTURE_2D, texid);
               }
```

```
        }
     else {
        update = 1;
        }
     }
   }
else {
  update = 1;
  s = wcr->patternname;
  slen = strlen(wcr->patternname);
  }

if (update) {
   /*
    * Allocate texture
    */
   int bmwidth, size, winsize;
   int ix, iy, i, j;
   unsigned char *img, *src, *dest;
   tended struct b_record *recp = rp;
   char data[MAXXOBJS];
   char *buf = data;
   C_integer v, bits[MAXXOBJS];
   int symbol;

   if ((slen > 0) && isdigit(s[0])) {
      /*
       * If the pattern starts with a number it is a width , bits encoding
       */
      nbits = MAXXOBJS;
      switch (parsepattern(s, slen, &width, &nbits, bits)) {
         case Failed:
        return Failed;
         case RunError:
            return RunError;
         }

      for(i = 0; i < nbits; i++) {
         v = bits[i];
         for(j=0; j<width; j+=8) {
            *buf++ = v;
            v >>= 8;
            }
         }
      index = get_tex_index(wd, 1);
      texptr = &(wd->texIds[index]);
      }
   else if ((symbol = si_s2i(siPatternSyms, s)) >= 0) {
      /*
       * Otherwise, it is a named pattern.  Find the symbol id.
       *
       * See if named pattern has been allocated already
       * If so, get texture id and move on
       */
      nbits = width = 8;
      if (wd->stdPatTexIds[symbol] > 0) {
         texid = wd->stdPatTexIds[symbol];
         }
      else {
         for(i = 0; i < 8; i++) {
            v = patbits[symbol * 8 + i];
            *buf++ = v;
            }
         texptr = &(wd->stdPatTexIds[symbol]);
         }
```

```
      }

   if (!texptr) { /* stdpat already allocated */
      UGLBindTexture(GL_TEXTURE_2D, texid);
      }
   else {
      /*
       * Convert from bits to bytes.
       *
       * Need to invert the pattern across the y-axis since OpenGL reads
       * from the lower left corner first (bottom to top, left to right)
       * from the array start to end.
       */
      size = 4*width*nbits;
      img = (unsigned char *)malloc(size);
      if (!img) return RunError;
      bitmap_to_pixmap(data, width, nbits, img, INVERT, LOW_TO_HIGH);

      /*
       * Generate a texture based off the pattern (img)
       */
      InitTexture2d(texptr);
      UGLTexImage2D(wd, GL_TEXTURE_2D, 0, GL_RGBA8, width, nbits, 0,
                     GL_RGBA, GL_UNSIGNED_BYTE, img, RunError);
      free(img);
      }

   /*
    * Update record fields
    */
   if (rp) {
      MakeInt(width, &(rp->fields[4]));
      MakeInt(nbits, &(rp->fields[5]));
      if (texptr)
         MakeInt(*texptr, &(rp->fields[6]));
      else
         MakeInt(texid, &(rp->fields[6]));
      /*
       * Index of 0 indicates it is a std pattern, otherwise it is the index
       * into the general texture id array (wd->texIds)
       */
      MakeInt(index, &(rp->fields[7]));
      }
   }

/*
 * Update render context pattern name
 */
if (rp) {
   if (wcr->patternname != NULL)
      free(wcr->patternname);
   wcr->patternname = malloc(slen+1);
   if (wcr->patternname == NULL) return RunError;
   strncpy(wcr->patternname, s, slen);
   wcr->patternname[slen] = '\0';
   }

/*
 * Apply pattern to stencil buffer by using texture and alpha test
 */
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
EnableStencilWrite(GL2D_PATT_BIT,0,GL_REPLACE);

RenderTexturedRect(w,0,0,ws->width,ws->height,width,nbits,-CNEAR);
```

```
/* restore */
DisableStencilWrite();
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);

UGLBindTexture(GL_TEXTURE_2D, 0); /* unbind */

return Succeeded;
}
```

Listing 4.26: The internal C function that renders **pattern** to the pattern bitplane

Clipping

OpenGL's scissor test was a perfect match for Unicon clipping. The scissor test, if enabled, defines a rectangular area in which rendering operations apply, but all operations outside of the specified area fail. When clipping is disabled, the scissor test is disabled. When a clipping area is defined, the arguments are parsed and passed to the OpenGL function **glScissor()**. The internal C function **setclip2d()** handles all clipping functionality.

```
int setclip2d(wbp w, struct b_record *rp)
   {
   wsp ws = w->window;
   wcp wcr = &(ws->wcrender);
   int clipx, clipy, clipw, cliph, px, py;
   double wx, wy;

   if (rp == NULL) {
      clipx = wcr->clipx;
      clipy = wcr->clipy;
      clipw = wcr->clipw;
      cliph = wcr->cliph;
      }
   else {
      GetInt(rp->fields[2], clipx);
      GetInt(rp->fields[3], clipy);
      GetInt(rp->fields[4], clipw);
      GetInt(rp->fields[5], cliph);

      /* update render context */
      wcr->clipx = clipx;
      wcr->clipy = clipy;
      wcr->clipw = clipw;
      wcr->cliph = cliph;
      }

   /* clipping disabled */
   if (clipx == 0 && clipy == 0 && clipw == -1 && cliph == -1) {
      if (glIsEnabled(GL_SCISSOR_TEST) == GL_TRUE)
         glDisable(GL_SCISSOR_TEST);
      }
   else {
```

```
      if (glIsEnabled(GL_SCISSOR_TEST) == GL_FALSE)
         glEnable(GL_SCISSOR_TEST);

      /* recook translation */
      clipx += wcr->dx;
      clipy += wcr->dy;
      glScissor(clipx, ws->height-(clipy+cliph), clipw, cliph);
      }
   return Succeeded;
   }
```

Listing 4.27: The implementation of clipping

Translation

The translational attributes **dx** and **dy** did not experience much revamping other than the process of removing translations coordinates from coordinates passed to the OpenGL implementation of the Unicon internal 2D API using macro **RemoveDxDy()**. The reasoning behind this design choice was to ensure that all coordinates on the 2D display list are pure so that the translational display list records can be modified by the end user to apply new translational coordinates to the display list records following. During render time, the translational coordinates of the Unicon window's render context are added back into the coordinates using macro **AddDxDy()**.

```
/*
 * Checks to see if (x, y, w, h) are the defaults set by rectargs(). If they
 * aren't, subtract dx/dy.
 */
#define RemoveDxDy(win, x, y, w, h) do {\
   wsp ws = win->window;\
   wcp wcr = &(ws->wcrender);\
   if (x != 0 || y != 0 || w != ws->width || h != ws->height) {\
      x -= wcr->dx;\
      y -= wcr->dy;\
      }\
   } while(0)

/*
 * Adds dx/dy back in via render context.
 * Uses rectargs() logic like RemoveDxDy()
 */
#define AddDxDy(win, x, y, w, h) do {\
   wsp ws = win->window;\
   wcp wcr = &(ws->wcrender);\
   if (x != 0 || y != 0 || w != ws->width || h != ws->height) {\
      x += wcr->dx;\
      y += wcr->dy;\
      }\
   } while(0)
```

Listing 4.28: Macros for removing and adding **dx** and **dy**

4.1.7 Graphical Primitives

Geometry

All Unicon geometric primitives (arcs, circles, lines, line segments, polygons, points, and rectangles) are rendered with the internal C function **drawgeometry2d()**. **drawgeometry2d()** not only renders but also initializes the arrays of vertices for each primitive type. As discussed for attribute linestyle, all unfilled primitive are initialized with a second array containing vertices for a segmented version of the original line. The initialization reoccurs whenever a graphical primitive has a user-modifiable field or an upstream translational coordinate field changed. The first element of each array of vertices (segmented or not) specifies the OpenGL draw code to be used.

This implementation uses immediate mode rendering with **glBegin()** and **glEnd()** which is encapsulated by the macro **RenderRealarray()**. **RenderRealarray()** is called from both **RenderRealarrayLinestyle()** and **RenderRealarrayFillstyle()** and takes an array of 2D vertices and its size. Another possible avenue for future work could be implementing vertex buffer objects to bypass OpenGL's fixed pipeline.

```
#define RenderRealarray(v, n) do {\
   glEnableClientState(GL_VERTEX_ARRAY);\
   glVertexPointer(2, GL_DOUBLE, 0, (void *) &(v)[1]);\
   glDrawArrays((v)[0], 0, (n-1)/2);\
   glDisableClientState(GL_VERTEX_ARRAY);\
   } while(0)
```

Listing 4.29: The base geometric rendering macro

One other item of note is that rendering with **GL_POLYGON** is only well-defined for simple convex polygons. Auxiliary internal C function **polygon_type** identifies if a polygon is complex (self-intersecting), simple convex, or non-convex. These are definitons used by X11. **polygon_type** uses an algorithm provided by Rory Daulton [9]. If filled polygon is not convex, then a two-pass algorithm using the stencil buffer from Chapter 14 of *OpenGL Programming Guide* [17] is used.

```
/*
 * Allocates and calculates coordinates for all geometric
 * primitives (polygons, circles, arcs, rectangles, lines, points,
 * line segments). Incorporates fillstyle for the primitives which
 * are filled.
 */
int drawgeometry2d(wbp w, struct b_record *rp)
   {
   wsp ws = w->window;
   wcp wc = w->context, wcr = &(ws->wcrender);
   double dx, dy, *v, *vseg, wx, wy;
   tended struct b_realarray *ap, *apseg;
   tended struct b_record *rp_t = rp;
   word n, nseg;
   word i, j, k;
   int rv, fill, polytype = 0, drawcode, intcode, recalc = 0;

   intcode = IntVal(rp->fields[1]);
   dx = wcr->dx;
   dy = wcr->dy;

   /*
    * Get the base types and determine whether coordinates need to be
    * recalculated
    */
   ...
   if (!is:list(rp->fields[2]) || recalc || ws->resize || RecalcTranslation(w))
      {
      /*
       * (Re)calculate array coordinates
       */
      switch (intcode) {
         case GL2D_FILLCIRCLE:
         case GL2D_FILLARC:
         case GL2D_DRAWCIRCLE:
         case GL2D_DRAWARC: {
            ...
            break;
            }

         case GL2D_FILLRECTANGLE:
         case GL2D_DRAWRECTANGLE: {
            ...
            break;
            }

         case GL2D_FILLPOLYGON:
         case GL2D_DRAWPOINT:
         case GL2D_DRAWPOLYGON:
         case GL2D_DRAWLINE:
         case GL2D_DRAWSEGMENT: {
            ...
            break;
            }
         default:
            return Failed;
         } /* end switch */
      } /* end if */
   else {
      /*
       * Get reference to array(s)
       */
      ...
      }
```

```
/*
 * Render primitive
 *
 * Linestyle and fillstyle are accounted for during this time for line
 * and filled primitives respectively. Points are the only primitive
 * that are rendered as is.
 */
glPushMatrix();
glTranslated(0,0,-CNEAR);
switch (intcode) {
   case GL2D_DRAWPOINT:
      RenderRealarray(v, n);
      break;

   case GL2D_FILLPOLYGON:
   case GL2D_FILLCIRCLE:
   case GL2D_FILLARC:
   case GL2D_FILLRECTANGLE:
      /*
       * For complex and nonconvex polygons:
       * Use a 2-pass stencil algorithm where writing to the color buffer is
       * disabled on the first pass to only draw to the stencil buffer.
       * Only regions drawn to an odd number of times (1st pass) will have a
       * '1' will be drawn to on the second pass.
       *
       * http://www.glprogramming.com/red/chapter14.html#name13
       */
      if (polytype == POLY_COMPLEX || polytype == POLY_NONCONVEX) {
         /*
          * First pass
          */
         EnableStencilWrite(GL2D_DRAW_BIT,0,GL_INVERT);
         glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
         RenderRealarray(v, n);
         glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
         DisableStencilWrite();

         /*
          * Second pass
          */
         RenderRealarrayFillstyle(w, v, n, GL2D_DRAW_BIT);
         }
      else {
         RenderRealarrayFillstyle(w, v, n, 0);
         }
      break;

   case GL2D_DRAWLINE:
   case GL2D_DRAWSEGMENT:
   case GL2D_DRAWPOLYGON:
   case GL2D_DRAWRECTANGLE:
   case GL2D_DRAWCIRCLE:
   case GL2D_DRAWARC:
      RenderRealarrayLinestyle(w, v, n, vseg, nseg);
      break;
   } /* end switch */
glPopMatrix();

return Succeeded;
}
```

Listing 4.30: Internal C function for rendering geometric primitives

Images

The remaining graphical primitives are images. To be more specific, these are 2D rectangular areas of pixels specified with RGB or RGBA values. These primitives, with exception of **EraseArea()**, are rendered as textured rectangles. Texturing is significantly faster than using **glDrawPixels()**.

The macros **RenderFilledRect()**, **RenderTexturedRect()**, **RenderTextured-BitmapRect()**, **_RenderFilledRect()** and **_RenderTexturedRect()** were written to encapsulate rendering for Unicon image primitives. The macros **RenderFilledRect()** and **_RenderFilledRect()** render a solid filled rectangle while **RenderTexturedRect()** and **_RenderTexturedRect()** render a textured rectangle by using the currently bound 2D texture. **RenderTexturedBitmapRect()** renders a bitmap, or bi-level image, as a textured rectangles with or without background fill. If background fill is desired, a filled rectangle is drawn with the current background color. The color and depth masks are disabled and the texture is rendered to the drawing bitplane of the stencil buffer using alpha testing. The color and depth masks are re-enabled and a final filled rectangle is drawn with the drawing bitplane of the stencil buffer.

```
#define _RenderFilledRect(wx1, wy1, wx2, wy2, near) do {\
   glBegin(GL_TRIANGLE_FAN);\
   glVertex3d((GLdouble)wx1,(GLdouble)wy1,(GLdouble)near);\
   glVertex3d((GLdouble)wx1,(GLdouble)wy2,(GLdouble)near);\
   glVertex3d((GLdouble)wx2,(GLdouble)wy2,(GLdouble)near);\
   glVertex3d((GLdouble)wx2,(GLdouble)wy1,(GLdouble)near);\
   glEnd();\
   } while(0)

#define RenderFilledRect(x, y, wd, ht, near) do {\
   double wx1, wy1, wx2, wy2;\
   wx1 = GLWORLDCOORD_X(w, x);\
   wy1 = GLWORLDCOORD_Y(w, y);\
   wx2 = GLWORLDCOORD_X(w, x+wd);\
   wy2 = GLWORLDCOORD_Y(w, y+ht);\
   _RenderFilledRect(wx1, wy1, wx2, wy2, near);\
   } while(0)

#define _RenderTexturedRect(w, x, y, wd, ht, texwd, texht, near) do {\
   wcp wcr = &(w->window->wcrender);\
   int drawop;\
   double wx1, wy1, wx2, wy2;\
   float tx1, ty1, tx2, ty2;\
\
   tx1 = ty2 = 0.0;\
```

```
   tx2 = (float)wd/(float)texwd;\
   ty1 = (float)ht/(float)texht;\
   wx1 = GLWORLDCOORD_X(w, x);\
   wy1 = GLWORLDCOORD_Y(w, y);\
   wx2 = GLWORLDCOORD_X(w, x+wd);\
   wy2 = GLWORLDCOORD_Y(w, y+ht);\
\
   glBegin(GL_TRIANGLE_FAN);\
   glTexCoord2f(tx1,ty1); glVertex3d(wx1,wy1,near);\
   glTexCoord2f(tx1,ty2); glVertex3d(wx1,wy2,near);\
   glTexCoord2f(tx2,ty2); glVertex3d(wx2,wy2,near);\
   glTexCoord2f(tx2,ty1); glVertex3d(wx2,wy1,near);\
   glEnd();\
   } while(0)

#define RenderTexturedRect(w, x, y, wd, ht, texwd, texht, near) do {\
   glEnable(GL_TEXTURE_2D);\
   glDisable(GL_TEXTURE_GEN_S);\
   glDisable(GL_TEXTURE_GEN_T);\
   _RenderTexturedRect(w,x,y,wd,ht,texwd,texht,near);\
   glEnable(GL_TEXTURE_GEN_S);\
   glEnable(GL_TEXTURE_GEN_T);\
   glDisable(GL_TEXTURE_2D);\
   } while(0)


#define FILL_BG 1
#define TRANSP_BG 0
#define RenderTexturedBitmapRect(w, x, y, wd, ht, texwd, texht, near, fillbg)\
   do {\
   wcp wcr = &(w->window->wcrender);\
   int drawop;\
   double wx1, wy1, wx2, wy2;\
   float tx1, ty1, tx2, ty2;\
\
   tx1 = ty2 = 0.0;\
   tx2 = (float)wd/(float)texwd;\
   ty1 = (float)ht/(float)texht;\
   wx1 = GLWORLDCOORD_X(w, x);\
   wy1 = GLWORLDCOORD_Y(w, y);\
   wx2 = GLWORLDCOORD_X(w, x+wd);\
   wy2 = GLWORLDCOORD_Y(w, y+ht);\
\
   if (fillbg) {\
      SetDrawopColorState(w, wcr->drawop, BG);\
      _RenderFilledRect(wx1, wy1, wx2, wy2, near);\
      SetDrawopColorState(w, wcr->drawop, FG);\
      }\
\
   glEnable(GL_TEXTURE_2D);\
   glDisable(GL_TEXTURE_GEN_S);\
   glDisable(GL_TEXTURE_GEN_T);\
\
   /* Apply pattern to stencil buffer by using texture and alpha test*/\
   glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);\
   glDepthMask(GL_FALSE);\
   EnableStencilWrite(GL2D_DRAW_BIT,0,GL_REPLACE);\
\
   /* first pass - render texture to stencil buffer */\
   glBegin(GL_TRIANGLE_FAN);\
   glTexCoord2f(tx1,ty1); glVertex3d(wx1,wy1,near);\
   glTexCoord2f(tx1,ty2); glVertex3d(wx1,wy2,near);\
   glTexCoord2f(tx2,ty2); glVertex3d(wx2,wy2,near);\
   glTexCoord2f(tx2,ty1); glVertex3d(wx2,wy1,near);\
   glEnd();\
```

```
\
   /* disable stencil writing */\
   DisableStencilWrite();\
   glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);\
   glDepthMask(GL_TRUE);\
\
   glEnable(GL_TEXTURE_GEN_S);\
   glEnable(GL_TEXTURE_GEN_T);\
   glDisable(GL_TEXTURE_2D);\
\
   /* second pass using stencil buffer */\
   SetStencilFunc(w, GL2D_DRAW_BIT);\
   _RenderFilledRect(wx1, wy1, wx2, wy2, near);\
   DefaultStencilFunc(w);\
   } while(0)
```

Listing 4.31: Macros for rendering image primitives

Each image primitive that uses a texture first has to allocate and initialize the texture using **get_tex_index()** and **InitTexture2d()**. **InitTexture2d()** is a macro that binds the texture with macro **UGLBindTexture()** (wrapper for **glBindTexture()**) and initializes the texture parameters. The texture is then loaded with the RGBA pixmap using **UGLTexImage2D()**, a macro wrapper for OpenGL function **glTexImage2D()** which checks for an OpenGL out-of-memory error. The **UGL** (Unicon OpenGL) prefix is reserved for macro wrappers of OpenGL functions that check for errors.

According to documentation, any OpenGL function could trigger an out-of-memory error, but **glTexImage2D()** is high on that list because it likely needs to allocate space to store the pixmap. If an out-of-memory error occurs, then the first half of the texture ID array **texIds** is deallocated and texture loading is attempted again. If the system is still out of memory, the program is terminated because some other issue is probably at hand. On subsequent redraws, if the saved texture ID and index match the texture ID held in the texture array, then there is no need to reallocate the texture. It is simply bound and rendered. Textures are bound to the 2D texture target using **UGLBindTexture()**, a macro wrapper for **glBindTexture()**.

```
#define UGLTexImage2D(wd, target, level, internalfmt, width, height, border,\
   format, type, data, retval) \
   do {\
   unsigned int err, errcount = 0;\
   glTexImage2D(target,level,internalfmt,width,height,border,format,type,data);\
   while ((err = glGetError()) != GL_NO_ERROR) {\
      if (err == GL_OUT_OF_MEMORY) {\
```

```
         if (!errcount) {\
            delete_first_tex(wd, (wd)->numTexIds/2);\
            glTexImage2D(target,level,internalfmt,width,height,border,format,\
              type,data);\
            }\
         /* if failed already, then let it die */\
         else return retval;\
         errcount++;\
         }\
      }\
   } while(0)
```

Listing 4.32: Macro wrapper for OpenGL texture loading function

Bi-level images The internal C function **drawblimage2d()** implements the rendering of bi-level images. The bitmap created from the Unicon string specification is used to create an RGBA alpha-based pixmap using **bitmap_to_pixmap()**. A texture is allocated and loaded with the pixmap and the bi-level image is rendered with **RenderTexturedBitmapRect()**.

```
int drawblimage2d(wbp w, struct b_record *rp)
   {
   wsp ws = w->window;
   wcp wcr = &(ws->wcrender);
   wdp wd = ws->display;
   char c, ch, *s;
   int width, height, len, update = 0;
   double x, y;
   unsigned char *bitmap;
   unsigned int texid, index, *texptr;

   if (!rp) return Failed;

   GetDouble(rp->fields[2], x);
   GetDouble(rp->fields[3], y);
   GetInt(rp->fields[4], width);
   GetInt(rp->fields[5], height);
   GetStr(rp->fields[6], s, len);
   ch = (char)IntVal(rp->fields[7]);
   bitmap = StrLoc(rp->fields[8]);

   if (is:integer(rp->fields[9]) && is:integer(rp->fields[10])) {
      texid = IntVal(rp->fields[9]);
      index = IntVal(rp->fields[10]);

      if (wd->texIds[index] != texid) {
         update = 1;
         }
      else {
         UGLBindTexture(GL_TEXTURE_2D, texid);
         }
      }
   else
      update = 1;

   if (update) {
```

```
tended struct b_record *rp_t = rp;
unsigned int ix, iy, bmwidth, index, shift, m, msk1, bytes;
unsigned char *tmp;

if (!bitmap) {

/* determine bitmap dimensions, each byte contains 8 pixel values */
bmwidth = width/8;
if (width \% 8)
   bmwidth++;
bytes = bmwidth*height;

Protect(bitmap = alcstr(NULL, bytes), return RunError);

/*
 * Read the image string and set the pixel values.  Note that
 * the hex digits in sequence fill the rows *right to left*.
 */
m = width \% 4;
if (m == 0)
   msk1 = 8;
else
   msk1 = 1 << (m - 1);              /* mask for first byte of row */

ix = width;
iy = 0;
m = msk1;
while (len--) {
   if (isxdigit(c = *s++)) {         /* if hexadecimal character */
  if (!isdigit(c))               /* fix bottom 4 bits if necessary */
   c += 9;
while (m > 0) {                /* set (usually) 4 pixel values */
   --ix;
         index = ix/8 + bmwidth*(height-(iy+1));
         shift = 7 - ix \% 8;
   if (c & m) {
            bitmap[index] |= 1 << shift; /* set */
      }
   else {
            bitmap[index] &= ~(1 << shift); /* clear */
      }
   m >>= 1;
   }
if (ix == 0) {                 /* if end of row */
   ix = width;
   iy++;
   m = msk1;
   }
else
   m = 8;
}
   }
if (ix > 0) {                        /* pad final row if incomplete */
   while (ix < width) {
      bitmap[ix/8 + bmwidth*(height-(iy+1))] &= ~(1 << (7 - ix \% 8)); /* clear */
      }
   }
MakeStr(bitmap, bytes, &(rp_t->fields[8]));
}

/* create texture */
tmp = malloc(4*width*height);
if (!tmp) return RunError;
bitmap_to_pixmap(bitmap, width, height, tmp, DONT_INVERT, HIGH_TO_LOW);
```

```
    index = get_tex_index(wd, 1);
    texptr = &(wd->texIds[index]);
    InitTexture2d(texptr);
    UGLTexImage2D(wd, GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA,
                  GL_UNSIGNED_BYTE, tmp, RunError);
    free(tmp);

    MakeInt(*texptr, &(rp->fields[9]));
    MakeInt(index, &(rp->fields[10]));
    }

  AddDxDy(w, x, y, width, height);

  if (ch == TCH1) /* transparent background */
    RenderTexturedBitmapRect(w,x,y,width,height,width,height,-CNEAR,
                             TRANSP_BG);
  else
    RenderTexturedBitmapRect(w,x,y,width,height,width,height,-CNEAR,FILL_BG);
  UGLBindTexture(GL_TEXTURE_2D, 0); /* unbind */
  return Succeeded;
  }
```

Listing 4.33: Internal C function for rendering bi-level images

String image The internal C function **drawstrimage2d()** contains the implementation of rendering a Unicon string image. First, the string image is converted to an RGBA pixmap and loaded into a newly allocated texture. It is then rendered using **RenderTexturedRect()**.

```
int drawstrimage2d(wbp w, struct b_record *rp)
  {
  wsp ws = w->window;
  wcp wcr = &(ws->wcrender);
  wdp wd = ws->display;
  int width, height;
  double x, y;
  unsigned char *pixmap;
  unsigned int texid, index, update = 0;

  if (!rp) return Failed;

  GetDouble(rp->fields[2], x);
  GetDouble(rp->fields[3], y);
  GetInt(rp->fields[4], width);
  GetInt(rp->fields[5], height);

  if (is:integer(rp->fields[7]) && is:integer(rp->fields[8])) {
    texid = IntVal(rp->fields[7]);
    index = IntVal(rp->fields[8]);
    if (wd->texIds[index] != texid)
      update = 1;
    else
      UGLBindTexture(GL_TEXTURE_2D, texid);
    }
  else {
    update = 1;
    }
```

```
/* (re)allocate texture */
if (update) {
   unsigned int *texptr;

   pixmap = StrLoc(rp->fields[6]);
   index = get_tex_index(wd, 1);
   texptr = &(wd->texIds[index]);
   InitTexture2d(texptr);
   UGLTexImage2D(wd, GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA,
                 GL_UNSIGNED_BYTE, pixmap, RunError);

   MakeInt(texid = *texptr, &(rp->fields[7]));
   MakeInt(index, &(rp->fields[8]));
   }

AddDxDy(w, x, y, width, height);
RenderTexturedRect(w,x,y,width,height,width,height,-CNEAR);
UGLBindTexture(GL_TEXTURE_2D, 0); /* unbind */
}
```

Listing 4.34: Internal C function for rendering String Images

Read image The internal C function **gl_readimage()** calls **gl_loadimage** to load an XBM or XPM file. The helper functions **load_xbm()** and **load_xpm()** were written to be platform neutral XBM/XPM file loaders that return a bitmap or pixmap respectively for use by the OpenGL implementation. It should be noted that **load_xbm()** treats all colors that do not match the background RGB values as the foreground color (bit set).

The image returned from **gl_loadimage** is rendered in **drawreadimage2d()** by loading it to a newly allocated texture and rendering it. If the image is a bitmap, **RenderTexturedBitmapRect()** is used. If the image is a pixmap, then **RenderTexturedRect()** is used.

```
int drawreadimage2d(wbp w, struct b_record *rp)
   {
   wsp ws = w->window;
   wcp wcr = &(ws->wcrender);
   wdp wd = ws->display;
   int width, height, is_pixmap, update = 0;
   double x, y;
   unsigned char *img;
   unsigned int texid, index;

   if (!rp) return Failed;

   GetDouble(rp->fields[2], x);
   GetDouble(rp->fields[3], y);
   GetInt(rp->fields[4], width);
   GetInt(rp->fields[5], height);
   is_pixmap = IntVal(rp->fields[6]);
```

```
if (is:integer(rp->fields[8]) && is:integer(rp->fields[9])) {
   texid = IntVal(rp->fields[8]);
   index = IntVal(rp->fields[9]);

   if (wd->texIds[index] != texid)
      update = 1;
   else
      UGLBindTexture(GL_TEXTURE_2D, texid);
   }
else
   update = 1;

if (update) {
   unsigned int *texptr;

   img = StrLoc(rp->fields[7]);
   index = get_tex_index(wd, 1);
   texptr = &(wd->texIds[index]);
   InitTexture2d(texptr);

   if (is_pixmap) {
      UGLTexImage2D(wd, GL_TEXTURE_2D, 0, GL_RGB8, width, height, 0, GL_RGB,
                    GL_UNSIGNED_BYTE, img, RunError);
      }
   else {
      unsigned char *tmp;

      tmp = malloc(4*width*height);
      if (!tmp) return RunError;
      bitmap_to_pixmap(img, width, height, tmp, DONT_INVERT, HIGH_TO_LOW);
      UGLTexImage2D(wd, GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0,
                    GL_RGBA, GL_UNSIGNED_BYTE, tmp, RunError);
      free(tmp);
      }

   MakeInt(texid = *texptr, &(rp->fields[8]));
   MakeInt(index, &(rp->fields[9]));
   }

AddDxDy(w, x, y, width, height);

if (is_pixmap) {
   RenderTexturedRect(w, x, y, width, height, width, height, -CNEAR);
   }
else { /* bitmap */
   RenderTexturedBitmapRect(w, x, y, width, height, width, height, -CNEAR,
                            TRANSP_BG);
   }

UGLBindTexture(GL_TEXTURE_2D, 0); /* unbind */
return Succeeded;
}
```

Listing 4.35: Internal C function for rendering images read from files

Font The internal C function **drawstringhelper()** handles font retrieval, storage, and rendering. To reduce overhead from retrieving font glyphs from FreeType, the internal C structure **struct fontsymbol** was created and added to the internal font structure

struct _wfont. Two additional fields, **library** and **face** were added as well to integrate FreeType. This implementation of FreeType only supports the set of extended ASCII characters. A possible avenue for future work is to add support for other character sets, such as unicode.

```
typedef struct _wfont {
   ...
#if HAVE_LIBFREETYPE
   FT_Library     library;
   FT_Face        face;
#endif                                    /* HAVE_LIBFREETYPE */
   struct fontsymbol chars[256];
   ...
} wfont, *wfp;

struct fontsymbol {
   unsigned char *pixmap;
   int width, height;
   int advance, top_bearing, left_bearing;
   unsigned int texid, index;
   };
```

Listing 4.36: Fields added to the internal font structure

drawstringhelper() performs both font rendering with or without background fill and returns the textwidth in pixels for the rendered font, as specified by argument **fill**. **drawstringhelper()** provides the option of only computing the textwidth with the argument **draw**.

The font rendering algorithm requires use of the OpenGL textures and stencil buffer. Each character of the text string is sequentially checked for whether it has been initialized within the internal font structure through field **chars**. The integer value of the character is the index into **chars**. To initialize a font symbol, a glyph is retrieved from FreeType. The auxiliary internal function **bitmap_to_pixmap()** converts the bitmap rasterized by FreeType into an alpha-based pixmap which is rendered to an OpenGL texture. The pixmap dimensions, relevant glyph metrics, and texture ID and index are stored into the respective slot in the font symbol array **chars**. Each character is then rendered via textured quad to the stencil buffer. After all characters are rendered, the stencil buffer is used to render the font drawn into it. If background fill is requested, a quad is drawn to the color buffer using the background color.

```
int drawstringhelper(wbp w, double x, double y, double z, char *s, int len,
   int fill, int draw)
   {
   wsp ws = w->window;
   wcp wcr = &(ws->wcrender);
   wdp wd = ws->display;
   wfp wf = wcr->font;
   int fheight, fdescent, textwidth;
   double startx, penx, peny;
   char *ptr = s;
   FT_Face face = wf->face;

   fheight = FT_FHEIGHT(face);
   fdescent = FT_DESCENT(face);
   penx = startx = x;
   peny = y + fdescent; /* center vertically around global font baseline */
   textwidth = 0;

   /* If rendering characters, apply settings */
   if (draw) {
      EnableStencilWrite(GL2D_DRAW_BIT,0,GL_REPLACE);
      glEnable(GL_TEXTURE_2D);
      glDisable(GL_TEXTURE_GEN_S);
      glDisable(GL_TEXTURE_GEN_T);
      }

   while (len--) {
      unsigned char c = *ptr++;
      struct fontsymbol *sym = &wf->chars[c];

      /* initialize character */
      if (!sym->pixmap) {
         FT_GlyphSlot glyph;
         FT_Bitmap ftbitmap;
         int size, rv, bmwidth;
         unsigned char *bitmap, *pixmap;

         /* Load a monochrome bitmap (1 bit per pixel) */
         rv = FT_Load_Char(face, c, FT_LOAD_RENDER | FT_LOAD_MONOCHROME);
         if (rv) {
            /* what to do in the case of a failure to load a glyph? */
            continue;
            }

         /* get glyph bitmap dimensions */
         glyph = wf->face->glyph;
         ftbitmap = glyph->bitmap;
         bitmap = ftbitmap.buffer;
         bmwidth = ftbitmap.pitch; /* # of bytes per row (incl. padding) */
         if (bmwidth < 0) bmwidth = -bmwidth;

         sym->height = ftbitmap.rows;
         sym->width = 8*bmwidth;
         sym->advance = glyph->advance.x >> 6;
         sym->top_bearing = glyph->bitmap_top;
         sym->left_bearing = glyph->bitmap_left;

         /* convert the bitmap to a pixmap (to be used as a texture) */
         size = 4*(sym->width)*(sym->height);
         sym->pixmap = (unsigned char *)malloc(size);
         if (!sym->pixmap) return RunError;
         bitmap_to_pixmap(bitmap, sym->width, sym->height, sym->pixmap, INVERT,
                          HIGH_TO_LOW);
         }
```

```
    if (draw) {
       double x1, y1;

       if (!sym->texid || wd->texIds[sym->index] != sym->texid) {
          unsigned int *texptr;

          /* allocate texture */
          sym->index = get_tex_index(wd, 1);
          texptr = &(wd->texIds[sym->index]);
          InitTexture2d(texptr);
          UGLTexImage2D(wd, GL_TEXTURE_2D, 0, GL_RGBA8, sym->width,
                        sym->height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                        sym->pixmap, RunError);
          sym->texid = *texptr;
          }
       else {
          UGLBindTexture(GL_TEXTURE_2D, sym->texid);
          }

       /*
        * Draw to stencil buffer
        *
        * Position character (low left corner) by using the left and top bearing
        * values. See FreeType glyph metrics.
        */
       x1 = penx + sym->left_bearing;
       y1 = peny - sym->top_bearing;
       _RenderTexturedRect(w,x1,y1,sym->width,sym->height,sym->width,
                           sym->height,z);
       penx += sym->advance;
       }

    textwidth += sym->advance;
    } /* end while */

 if (draw) { /* render text */
    glDisable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    DisableStencilWrite(); /* done writing to stencil buffer */
    UGLBindTexture(GL_TEXTURE_2D, 0); /* unbind */

    if (fill) { /* background fill */
       SetDrawopColorState(w, wcr->drawop, BG);
       RenderFilledRect(startx,y-fheight,textwidth,fheight,z);
       SetDrawopColorState(w, wcr->drawop, FG);
       }

    /* draw text */
    SetStencilFunc(w, GL2D_DRAW_BIT);
    RenderFilledRect(startx,y-fheight,textwidth,fheight,z);
    DefaultStencilFunc(w);
    }
 return textwidth;
 }
```

Listing 4.37: Internal C function for rendering font glyphs

Copyarea The internal C function **copyarea2d()** implements both the 2D and 3D semantics for Unicon graphics function **CopyArea()**. The source rectangle is read with

glReadPixels() into an RGBA pixmap. For both 2D and integrated 2D/3D modes, any off-screen pixels are set to the background color of the Unicon render context. If the window is operating in integrated 2D/3D mode, then a second pixmap grabs the depth buffer values with **glReadPixels()**. The depth value of each on-screen pixel is checked. If the depth value is greater than 0, meaning it is a 3D pixel, then set the alpha value of that pixel to 0. This ensures that only HUD, or 2D pixels, are copied with **CopyArea()** during 2D rendermode of the integrated 2D/3D windowing mode. The resulting pixmap, like the previous image primitives, are loaded to a newly allocated texture and rendered with **RenderTexturedRect()**. If **CopyArea()** is called with dimensions that fill the entire window, the entire display list is deleted and the current Unicon render context is saved as the default.

```
/*
 * 2d CopyArea() semantics (!ws->is_3D)
 *    Copies all pixels (same as legacy semantics)
 *
 * 2d/3d CopyArea() semantics (ws->is_3D):
 *    Only copies 2D/HUD pixels
 */
#define OffscreenPixel(ws, x, y, ix, iy) \
   ((real_iy+y < 0 || real_iy+y > (ws)->height || ix+x < 0 ||\
    ix+x > (ws)->width))
int copyarea2d(wbp w2, struct b_record *rp)
   {
   wbp w, tmp1, tmp2;
   wsp ws2 = w2->window, ws;
   wcp wcr2 = &(ws2->wcrender), wcr;
   wdp wd;
   double wx, wy, dx, dy;
   double x, y, x2, y2;
   int width, height, drawop, update = 0;
   unsigned char *depth, *pixmap = NULL;
   unsigned int texid, index, *texptr;

   if (!rp) return Failed;

   /* get attributes */
   if (!is:file(rp->fields[2]))
      w = NULL;
   else {
      w = BlkD(rp->fields[2],File)->fd.wb;
      if (!w || !w->refcount || !w->window || !w->context) {
         w = NULL;
         rp->fields[2] = nulldesc;
         }
      else {
         /* check if w is still valid */
         for (tmp1 = wbndngs, tmp2 = NULL; tmp1; tmp1 = tmp1->next) {
            if (tmp1 == w) {
               tmp2 = w;
```

```
            break;
            }
         }
      /* window (w) has been freed, make nulldesc */
      if (!tmp2) {
         w = NULL;
         rp->fields[2] = nulldesc;
         }
      }
   }

GetDouble(rp->fields[3], x);
GetDouble(rp->fields[4], y);
GetInt(rp->fields[5], width);
GetInt(rp->fields[6], height);
GetDouble(rp->fields[7], x2);
GetDouble(rp->fields[8], y2);

/*
 * Grab cooked values
 */
if (is:integer(rp->fields[9]) && is:integer(rp->fields[10])) {
   texid = (unsigned int)IntVal(rp->fields[9]);
   index = (unsigned int)IntVal(rp->fields[10]);

   if (w) {
      wd = w->window->display;
      }
   else {
      /* Since window is closed, the binding has been freed */
      rp->fields[2] = nulldesc;

      /*
       * if window is closed, assume same display
       * TODO: expand to encompass different displays
       */
      wd = ws2->display;
      }

   /* check if texture is still alive */
   if (wd->texIds[index] == texid)
      UGLBindTexture(GL_TEXTURE_2D, texid);
   else
      update = 1;
   }
else if (w) {
   wd = w->window->display;
   if (wd != ws2->display) {
      return Failed;
      }
   update = 1;
   }
else /* window closed before read... */
   return Failed;

/*
 * Get the rectangular area from {w}
 */
if (update) {
   tended struct b_record *rp_t = rp;
   int ix, iy, px, py;
   unsigned char bg[4];
   int outofbounds = 0, size, drawop;

   ws = w->window;
```

```
/*
 * Get area from source window
 */
MakeCurrent(w);
AddDxDy(w, x, y, width, height);

/* check for out-of-bounds rectangle */
if ((x+width > ws->width || x < 0) || (y+height > ws->height || y < 0)) {
   outofbounds = 1;
   }

px = x;
py = ws->height - (y + height); /* correct for OpenGL conventions */

/* get color buffer */
size = width*height;
pixmap = (unsigned char *)malloc(4*size);
if (!pixmap) return RunError;
glReadPixels(px,py,width,height,GL_RGBA,GL_UNSIGNED_BYTE,pixmap);

if (ws->is_3D) {
   /* get depth buffer */
   depth = (unsigned char *)malloc(size);
   if (!depth) return RunError;
   glReadPixels(px,py,width,height,GL_DEPTH_COMPONENT,GL_UNSIGNED_BYTE,
                depth);
   for (iy = 0; iy < height; iy++) {
      int real_iy = height-iy-1; /* unicon y-coord */
      for (ix = 0; ix < width; ix++) {
         /* make off-screen pixels transparent */
         if (outofbounds && OffscreenPixel(ws,x,y,ix,real_iy)) {
            pixmap[4*(iy*width+ix)+3] = 0;
            }
         /* make 3D pixels transparent */
         else if (depth[iy*width+ix] != 0) {
            pixmap[4*(iy*width+ix)+3] = 0;
            }
         }
      }
   free(depth);
   }
/*
 * Fill offscreen pixels with bg color
 */
else if (outofbounds) { /* !ws->is_3D */
   GetContextColorUC(w, BG, bg[0], bg[1], bg[2], bg[3]);
   for (iy = 0; iy < height; iy++) {
      int real_iy = height-iy-1; /* unicon y-coord */
      for (ix = 0; ix < width; ix++) {
         /* make off-screen pixels transparent */
         if (OffscreenPixel(ws,x,y,ix,real_iy)) {
            int index = 4*(iy*width+ix);
            AssignRGBA(&(pixmap[index]),bg[0],bg[1],bg[2],bg[3]);
            }
         }
      }
   }
MakeCurrent(w2); /* restore glX context */

/* create texture */
index = get_tex_index(wd, 1);
texptr = &(wd->texIds[index]);
InitTexture2d(texptr);
UGLTexImage2D(wd, GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA,
```

```
                GL_UNSIGNED_BYTE, pixmap, RunError);
   free(pixmap);

   /* Update field values */
   MakeInt(*texptr, &(rp_t->fields[9]));
   MakeInt(index, &(rp_t->fields[10]));
   }

AddDxDy(w2, x2, y2, width, height);
RenderTexturedRect(w2,x2,y2,width,height,width,height,-CNEAR);
UGLBindTexture(GL_TEXTURE_2D, 0); /* unbind */

return Succeeded;
}
```

Listing 4.38: Internal C function for copying rectangular areas

Erasearea The internal C function **erasearea2d()** is unlike the other image primitives in that it does not use textures. For 2D mode, it only needs to wipe a rectangular area to the current background color. This becomes trickier while in integrated 2D/3D mode. The rectangular area should clear all pixels in the area to reveal any drawn 3D pixels in the background that were obscured. The new addition of transparency to the 2D facilities complicates the implementation of **EraseArea()** integrated 2D/3D semantics because transparent objects must be drawn from far-to-near order to ensure correct color blending. Consequently, the 3D display list must be rendered before the 2D display list.

For integrated 2D/3D mode, **erasearea2d()** does nothing on the initial call. When it is called during display list traversal, it sets the erase bitplane of the stencil buffer and unsets the specified rectangular area to prohibit rendering. See Listing 4.6 in Section 4.1.3 for an explanation of the integrated 2D/3D mode traversal algorithm and Listing 4.6 located in the same section for traversal code. Like **CopyArea()**, if **EraseArea()** is called with arguments that would erase the entire screen, the display list is deleted and the current render context is saved as the default render context. Additionally, the screen is erased in accordance to the semantics of the window mode used.

```
int erasearea2d(wbp w, struct b_record *rp)
   {
   wsp ws = w->window;
   wcp wc = w->context, wcr = &(ws->wcrender);
   double x, y, width, height;
   double x1, y1, x2, y2;
```

```
if (!rp) return Failed;

GetDouble(rp->fields[2], x);
GetDouble(rp->fields[3], y);
GetDouble(rp->fields[4], width);
GetDouble(rp->fields[5], height);

AddDxDy(w, x, y, width, height);

if (ws->is_3D) {
   /* this should only run during a 3d's windows traversal redraw */
   glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
   glDepthMask(GL_FALSE);
   EnableStencilWrite(GL2D_ERASE_BIT,0xFF,GL_ZERO);
   RenderFilledRect(x, y, width, height, -CNEAR);
   DisableStencilWrite();
   glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
   glDepthMask(GL_TRUE);
   }
else {
   /*
    * Draw bg color to the far plane of the viewing volumes. This
    * erases all primitives in the rectangular area. 3d primitives
    * are drawn after 2d primitives to make this work
    */
   SetDrawopColorState(w, wcr->drawop, BG);
   glDepthFunc(GL_ALWAYS); /* enable writing always (regardless of depth) */
   DepthRangeMax; /* set depth range to max */
   RenderFilledRect(x,y,width,height,-CNEAR);
   DepthRange2d; /* restore depth range */
   glDepthFunc(GL_LEQUAL); /* restore default */
   SetDrawopColorState(w, wcr->drawop, FG);
   }
return Succeeded;
}
```

Listing 4.39: Internal C function for erasing rectangular areas

4.2 Integrated 2D/3D Facilities

The integrated graphics facilities are implemented as an extension to the existing 3D facilities. In terms of graphics mode cohesiveness, this decision makes the most sense. With 3D rendering, a HUD is just a 2D scene. Thus, 2D graphics rendering in the form of a HUD is a convenient extension to the current 3D graphics facilities.

4.2.1 Viewing Volume Sectioning

The Unicon 3D graphics facilities share OpenGL's right-handed coordinate system while the Unicon 2D graphics facilities uses screen coordinates with the origin at the top

left corner of the screen. The integrated facilities combine the semantics of each mode into a single mode with a switching mechanism.

Four macros have been added to keep track of the OpenGL viewing volume attributes, **CWIDTH()**, **CHEIGHT**, **CNEAR**, and **CFAR**. Two additional macros, **HALF_CWIDTH** and **HALF_CHEIGHT()**, were added for convenience. They are expressed in terms of OpenGL's 3D coordinate system and relative to the camera's position. The most important of these attributes is the z-value for the near plane of the viewing volume. The implementation of the Unicon 2D graphics facilities as a heads-up display hinges on reserving the near plane and a small buffer volume for rendering. This was done with the OpenGL function **glDepthRange()**, with valid depths between 0.0 and 1.0 inclusive. The macro wrapper **DepthRange2d()** specifies all drawn objects are to be at a depth of 0.0 while **DepthRange3d()** specifies all drawn objects to be between depths 0.05 and 1.0 inclusive.

```
#define CWIDTH 0.25
#define CNEAR 0.25
#define CFAR 50000.0
#define CHEIGHT(w) \
   (CWIDTH*((double)w->window->height)/((double)w->window->width))

#define HALF_CWIDTH (CWIDTH/2.0)
#define HALF_CHEIGHT(w) (CHEIGHT(w)/2.0)

#define DepthRange2d() glDepthRange(0.0,0.0)
#define DepthRange3d() glDepthRange(0.05,1.0)
#define DepthRangeMax() glDepthRange(1.0,1.0)
```

Listing 4.40: Viewing volume related macros

The macros **SwitchMode2d()** and **SwitchMode3d()** set the corresponding depth ranges and rendering states. The new internal context field **rendermode** is linked to the new Unicon context attribute **rendermode**. **"rendermode=2d"** specifies 2D render mode while **"rendermode=3d"** specifies 3D render mode. The internal canvas mirrors the addition of **rendermode** as a field. This internal canvas field indicates the current OpenGL rendermode state, i.e. whether the current OpenGL state is set up for 2D or 3D rendering. The macros **ApplyRendermode()** applies the specified rendermode by setting the internal canvas render state **rendermode** and calling the appropriate

SwitchMode*() macro. The **CheckRendermode()** macro is called in all 2D and 3D rendering functions before render occurs. It compares the context and canvas rendermodes and applies the context rendermode to the canvas if they differ.

```
#define SwitchMode2d(w) do {\
   if (IS_CLIP(w))\
      glEnable(GL_SCISSOR_TEST);\
   DepthRange2d();\
   } while(0)

#define SwitchMode3d(w) do {\
   if (IS_CLIP(w))\
      glDisable(GL_SCISSOR_TEST);\
   DepthRange3d();\
   } while(0)

#define UGL2D 0
#define UGL3D 1
#define ApplyRendermode(w,is_3d) do {\
   (w)->window->render_3d = is_3d;\
   if (is_3d)\
      SwitchMode3d(w);\
   else\
      SwitchMode2d(w);\
   } while(0)

#define CheckRendermode(w) do {\
   if ((w)->window->render_3d != (w)->context->render_3d) {\
      ApplyRendermode(w, w->context->render_3d);\
      }\
   } while(0)
```

Listing 4.41: Macros for apply rendermode changes

The 3D display list is rendered first followed by transforming the modelview matrix to the position of the camera to render the 2D display list. The default state of OpenGL's modelview matrix is at the camera's position from using **glFrustum()**. A **glPushMatrix()** followed by a **glLoadIdentity()** can be used to return to this default state without losing the current matrix. This implementation of the 2D facilities does not use lighting and all 2D rendering occurs with OpenGL lighting disabled.

```
int traversefunctionlist(wbp w)
   {
   wsp ws = w->window;
   int rendermode = ws->rendermode;

   /* Render 3d scene */
   if (is:list(ws->funclist)) {
      ApplyRendermode(w, UGL3D);

      if (traversefunclist(w) == Failed) { /* 3d */
         ApplyRendermode(w, rendermode);
         return Failed;
```

```
        }
    }

/* Render 2d scene */
if (is:list(ws->funclist2d)) {
    ApplyRendermode(w, UGL2D);
    glDisable(GL_LIGHTING);
    glPushMatrix();
    glLoadIdentity();

    if (traversefunclist2d(w) == Failed) { /* 2d */
        ApplyRendermode(w, rendermode);
        return Failed;
        }
    glPopMatrix();
    glEnable(GL_LIGHTING);
    }

/* restore settings */
ApplyRendermode(w, rendermode);

if (ws->resize) ws->resize = 0;

/* update screen */
FlushWindow(w);
return Succeeded;
}
```

Listing 4.42: The algorithm for rendering 2D and 3D display lists

The last problem is how to convert Unicon 2D screen coordinates to OpenGL world coordinates (OpenGL uses a right-handed coordinate system). OpenGL's modelview matrix stack saves the camera position and the internal canvas keeps track of the viewing volume dimensions. The crucial parameter, **cnear**, is a measure of the distance from the camera to the near plane (see Figure 4.2). Consequently, this makes the coordinate transformation a mapping between two 2D coordinate systems consisting of a scaling and a translation. This mapping is achieved by using the ratios of the window to viewing volume width and window to viewing volume height respectively. Another caveat is that OpenGL's coordinate system requires that the y-coordinate be converted to simulate an origin at the center of the screen. The coordinate transformation can be described by the two equations

$$x_{OpenGL} = x_{Unicon} * \frac{cwidth}{width} - \frac{cwidth}{2}$$
$$y_{OpenGL} = (height - y_{Unicon}) * \frac{cheight}{height} - \frac{cheight}{2}$$

the parameters of which can be found in Figure 4.2.

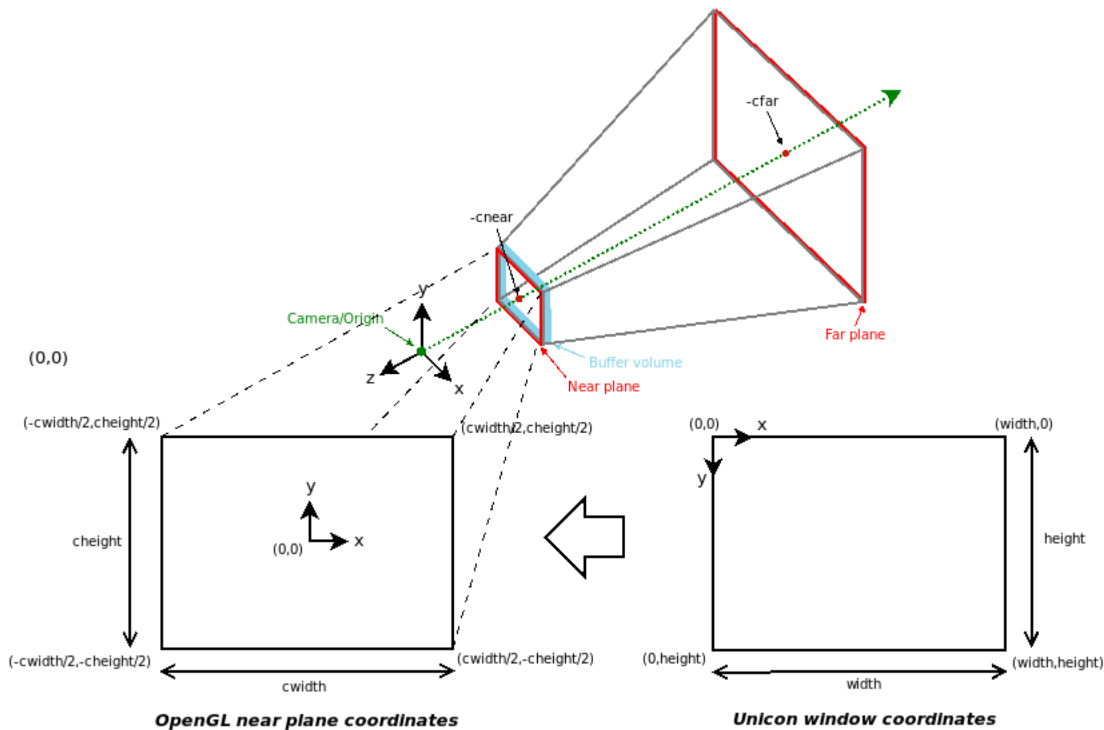

Figure 4.2: Unicon to OpenGL coordinate transformation

Four macros were written to implement the mapping of Unicon screen coordinates to OpenGL world coordinates: **GLWORLDCOORD_X()**, **GLWORLDCOORD_Y()**, **GLWORLDCOORD_RENDER_X()**, and **GLWORLDCOORD_RENDER_Y()**. The difference between these two sets of macro is that **GLWORLDCOORD*()** is used for transforming image primitive coordinates and **GLWORLDCOORD_RENDER*()** is used for geometric primitives. This is due to the diamond-rule with line, or geometric, based rendering. The start and endpoints need to land in the center of a pixel to achieve better accuracy. However, the diamond rule does not apply to pixel-based operations.

```
#define GLWORLDCOORD_X(w, px) ((px)*CLIPW(w)/(double)PIXW(w)-CLIPW(w)/2.0)
#define GLWORLDCOORD_Y(w, py) \
    ((PIXH(w)-(py))*CLIPH(w)/(double)PIXH(w)-CLIPH(w)/2.0)
#define GLWORLDCOORD_RENDER_X(w, px) \
    ((px+0.5)*CLIPW(w)/(double)PIXW(w)-CLIPW(w)/2.0)
#define GLWORLDCOORD_RENDER_Y(w, py) \
    ((PIXH(w)-(py+0.5))*CLIPH(w)/(double)PIXH(w)-CLIPH(w)/2.0)
```

Listing 4.43: Coordinate transformation macros

Chapter 5: Testing and Evaluation

The goal of this chapter is to determine the feature-completeness and release readiness of this OpenGL implementation of Unicon's 2D and integrated 2D/3D graphics facilities. There will be two main types of evaluation performed, qualitative and quantitative. Qualitative evaluation will determine the graphical degree of accuracy the OpenGL implementation achieves compared to the legacy Xlib implementation while quantitative evaluation will compare their render speeds.

Two 64-bit machines running Ubuntu 18.04.4 LTS (5.3.0-46-generic kernel) were used to run these tests to gauge the effect of the GPU on the performance of this implementation. Unfortunately, the non-GPU hardware specs are not identical, so the differing performances cannot be attributed to solely the difference in GPUs. Both machines have hardware acceleration and direct rendering enabled.

	Machine 1	Machine 2
CPU	Quad Core Intel Core i7-8565U	AMD Ryzen Threadripper 1950X (16-core)
GPU	Mesa Intel UHD Graphics 620	GeForce GTX 1070 Ti
Driver	v4.6 Mesa 20.1.0-devel (i915)	nvidia-435
Memory	15666.8MB	32025.9MB

Table 5.1: Testing machine hardware specifications

5.1 Qualitative Evaluation

The following qualitative comparisons will be between the Xlib and OpenGL implementations on machine 1 (integrated GPU). Any rendering differences between machines 1 and 2 due to graphics driver implementation will be pointed out, otherwise the graphical output of machines 1 and 2 are the same.

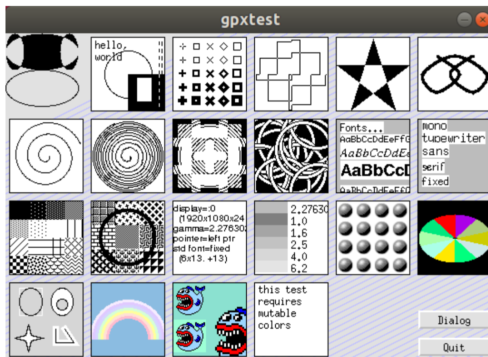

Figure 5.1: gpxtest (Xlib)

Figure 5.2: gpxtest (Mesa)

Figure 5.3: gpxtest dialog (Xlib)

Figure 5.4: gpxtest dialog (Mesa)

5.1.1 GPXTEST

The **gpxtest** program is a Unicon 2D graphics program that tests most, if not all, of the base capabilities offered by the 2D facilities. It also includes a small test of Icon's legacy GUI library.

Overall, the OpenGL implementation runs **gpxtest** quite well (see Figures 5.1 and 5.2). The most significant difference is that the OpenGL implementation supports mutable colors, while the legacy implementation does not. The fonts are different, but that is not a significant problem: each Unicon 2D implementation uses different fonts, although four portable font names are required to be present in some form. The most noticeable detriment of the OpenGL implementation is that thick lines (**linewidth** greater than 1) are not capped at all. Furthermore, thick curved lines do not have a uniform thickness. This is a possible avenue for future work, as mentioned back in the **linewidth** section.

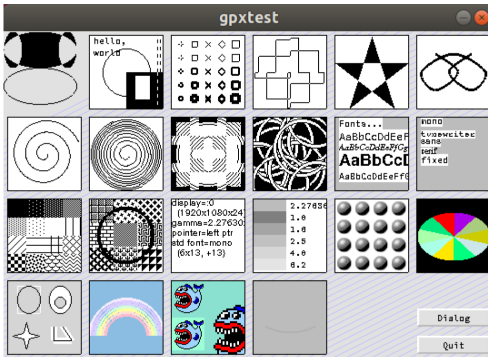

Figure 5.5: gpxtest (Mesa)

Figure 5.6: gpxtest (Nvidia)

Figure 5.7: gpxtest dialog (Mesa)

Figure 5.8: gpxtest dialog (Nvidia)

One other difference is that the **drawop** test looks slightly different (1st row, 4th column). The dialog window looks good as well, with comparisons about differing font and thick lines holding true (see Figures 5.3 and 5.4).

There are a few subtle differences between the Mesa and Nvidia drivers for the OpenGL implementation (see Figures 5.5 and 5.6). On some of the test rectangles, there is a thin white line on the implementation running on the Nvidia driver. This appears to be some sort of rounding error that does not occur on the Mesa driver. Perhaps a consequence of rounding errors is that the small crosses, boxes, and diamonds with varying linewidth look different as well. Unfortunately, these differences are due to driver implementation. The dialog window looks about the same except for the previously mentioned differences in thick lines (see Figures 5.7 and 5.8).

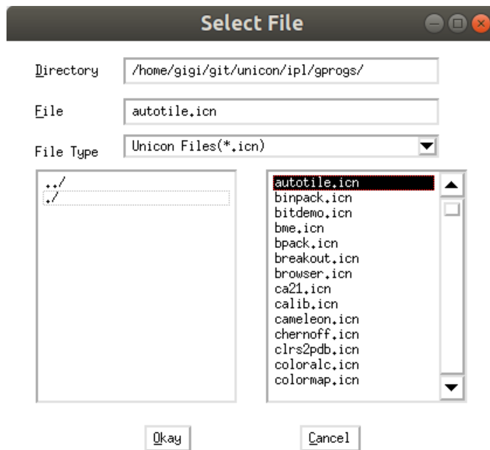

Figure 5.9: UI file selection (Xlib)

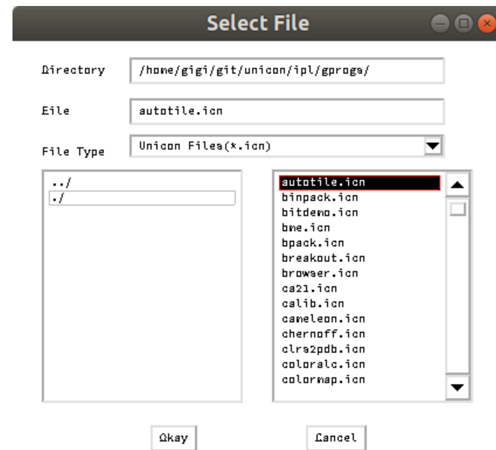

Figure 5.10: UI file selection (Mesa)

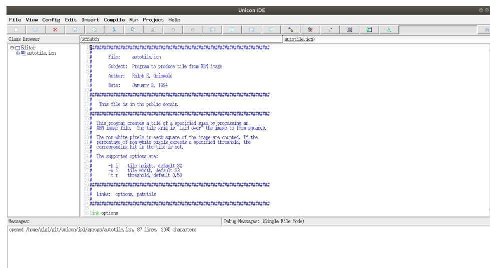

Figure 5.11: UI (Xlib)

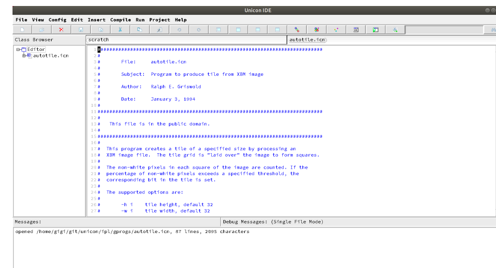

Figure 5.12: UI (Mesa)

5.1.2 UI

UI, the Unicon IDE, is built solely with the GUI class library. Font differences aside, the Xlib and OpenGL implementations look and run very similarly (see Figures 5.11 and 5.12). One subtle difference is that in the file selection window, the outline highlight for the Xlib implementation is finely dashed while the OpenGL version is solid (see Figures 5.9 and 5.10). The only differences between the OpenGL drivers is that, again, the Nvidia driver appears to have a rounding error such that highlighting in the file selection dialog is off (see Figures 5.13 and 5.14).

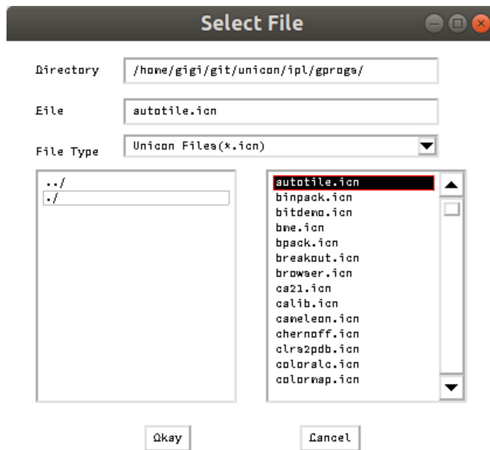

Figure 5.13: UI file selection (Mesa)

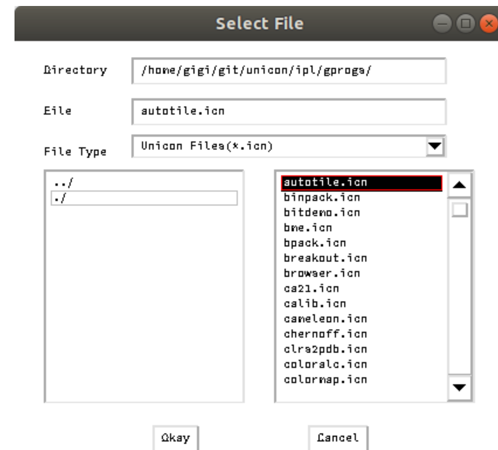

Figure 5.14: UI file selection (Nvidia)

5.1.3 IVIB

IVIB is a visual interface builder written with the GUI class library. Font differences aside, IVIB runs as expected on the OpenGL implementation as can be seen in Figures 5.15 and 5.16. The only noticeable difference is that item dragging is slightly slower on the OpenGL implementation.

The GUI class library buffers all rendering on a Unicon hidden window. As a result, the display list size of the IVIB window is inconsequential while the buffered, hidden window has a display list size of 170. On the other hand, the hidden window of UI has an intial display list size of 1100. On window resizes, the IVIB hidden window display list grows by 170 while the UI hidden window display list grows by 370. The display list will also grow through usual program use. UI is the largest real-world GUI application tested on the OpenGL implementation by display list size.

5.1.4 SPEEDTEST

speedtest is a Unicon program written to evaluate the performance of the Unicon 2D facilities. However, it is also important to evaluate the qualitative aspect of the **speedtest**. This program renders a selected number of a particular graphical primitive.

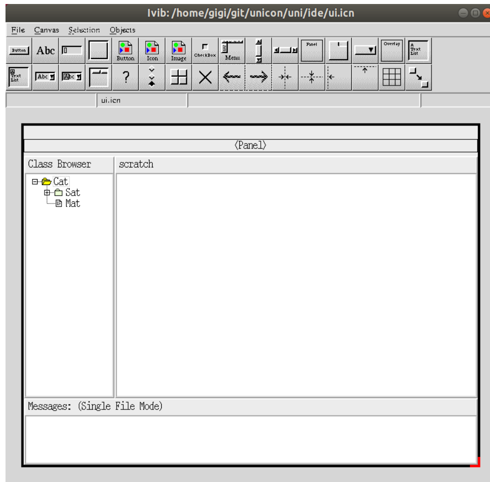

Figure 5.15: IVIB: loading a layout (Xlib)

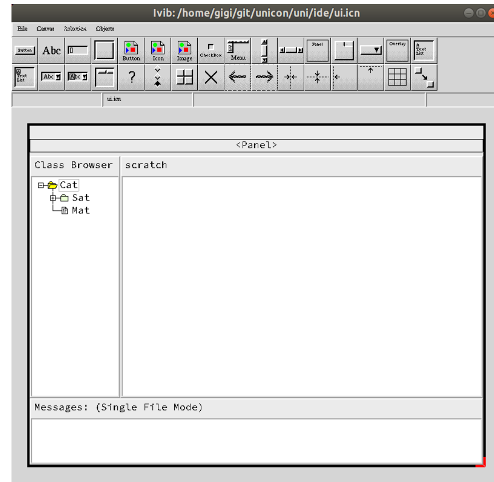

Figure 5.16: IVIB: loading a layout (Mesa)

The size, position, color, and if applicable, **linestyle** and **fillstyle** are randomized for each primitive drawn. This qualitative test renders 100,000 star polygons (from **gpxtest**) with **"fillstyle=masked"**. Figures 5.17 and 5.18 look identical. The Nvidia driver powered **speedtest** is identical as well.

Figure 5.17: speedtest: masked star polygons (Xlib)

Figure 5.18: speedtest: masked star polygons (Mesa)

5.1.5 JEB1

The **jeb1** demo is a precursor to CVE, a collaborative virtual environment. A goal of this thesis was to integrate 2D and 3D rendering into a window without having to mess with subwindows. This evaluation will compare the qualitative aspects of the legacy **jeb1** demo and show the fruits of this thesis project with a modified, or integrated, **jeb1** demo. The legacy **jeb1** demo runs well on the OpenGL implementation and is pretty much identical to the Xlib implementation apart from the font differences (see Figures 5.19 and 5.20).

Figure 5.19: Legacy jeb1 demo (Xlib)

Figure 5.20: Legacy jeb1 demo (Mesa)

The integration 2D/3D version of the **jeb1** demo dispenses with the 3D subwindow. Instead, the entire window is a 3D window (mode **"gl"**) with a transparent HUD which models the interface of its legacy counterpart (see Figure 5.21). However, there are incompatibilities between the integrated graphics mode and the GUI class library used for the HUD elements. If immediate mode rendering, **"buffer=off"**, is used, then the HUD flickers because the 2D elements must be removed each time to prevent bleeding with the transparent HUD. If double-buffering is used, then the menu items appear to not respond and render as they should, leaving artifacts in some cases (see Figure 5.22).

Figure 5.21: Integrated 2D/3D jeb1 demo

Figure 5.22: Integrated 2D/3D jeb1 demo rendering artifact

All display events by the GUI dispatcher are handled on a per-component basis, i.e. the parent integrated 2D/3D mode dialog is not guaranteed to get issued a redraw when one of its components receives an event and is redrawn. If a niche workaround with the GUI class library exists, it is neither easy nor intuitive. Instead, the GUI toolkit should be modifed to to support the OpenGL's double-buffered mode. It is not an option to use immediate mode (single-buffered) because it causes the HUD elements to flicker. The **About** notice dialog of the integrated **jeb1** demo flickers because it is not double-bufferd.

5.2 Quantitative Evaluation

Two Unicon programs, **speedtest** and **animation**, were written to evaluate the OpenGL implementation's performance speed. Source code for both programs can be found in Appendix C.

5.2.1 speedtest

The purpose of **speedtest** is to determine how the OpenGL implementation holds up to the Xlib implementation performance-wise. It does not matter if an implementation is graphically accurate if the programs run too slowly. The primitives chosen for this

comparison are filled and unfilled rectangles, arcs, circles, star polygons, text strings and copyareas. The text string that was rendered is **"This is a string!"**. Machine 1 was used to determine the number of primitives to be rendered for these tests. The number chosen was 100,000. This was reached by comparing the ratios of time-to-completion of each implementation for 10 to 1 million primitives, by orders of magnitude.

It was tricky to compare the ratios, because Xlib excels at rendering some primitives but is slow for others, i.e. the range of Xlib render times was much greater than OpenGL's. Xlib was fast at copyarea and filled primitives and regular line primitives but slow at rendering any sort of dashed line. The only outlier in terms of render time for OpenGL is copyarea, which is likely due to the synchronization bottleneck caused by **glReadPixels()**. As the number of primitives drawn increases, the differences between the extremes becomes greater. Consequently, 100,000 was chosen as a good middle ground.

Due to the variations in performances between different primitive types (see Tables 5.2 and 5.3), the average primitive render time was calculated for both machines. If each primitive tested here occurred equally often in a graphics program rendering 100,000 primitives, then the OpenGL implementation would be 2.029 times faster than the Xlib implementation on machine 1 and 1.695 times faster on machine 2 (see Table 5.4). The Xlib implementation runs 5.840 times faster on machine 2 than on machine 1 and the OpenGL implementation runs 4.877 times faster on machine 2 than on machine 1.

However, if the primitives in the program heavily favor filled and unfilled rectangles, arcs, circles, polygons, and copyareas, then the Xlib implementation will faster than the OpenGL implementation for both machines. Primitives with **linestyle** and **fillstyle** likely do not constitute a majority of drawing operations. If an equal number of filled and unfilled rectangles, arcs, circles, polygons, text, and copyareas were rendered in a program for rendering 100,000 primitives, then the OpenGL implementation would take 0.228 of the time that the Xlib implementation would on machine 1 and 0.277 of the time on machine 2. In other words, the Xlib implementation is 4.386 times faster than the

	Xlib (sec)	OpenGL (sec)	Xlib-to-OpenGL ratio
Filled rectangle	1.100	11.306	.0972
Filled masked rectangle	29.356	30.936	.9489
Filled textured rectangle	34.648	32.552	1.0643
Rectangle	6.172	13.784	.4477
Dashed rectangle	35.770	13.872	2.5785
Striped rectangle	70.186	14.972	4.6878
Filled arc	1.432	16.810	.0851
Filled masked arc	18.136	38.402	.4722
Filled textured arc	30.008	42.054	.7135
Arc	2.964	13.726	.2159
Dashed arc	263.036	13.354	19.6971
Striped arc	423.482	14.678	28.8514
Filled circle	2.222	19.326	.1098
Filled masked circle	12.062	42.410	.2844
Filled textured circle	25.186	44.826	.5618
Circle	0.976	13.470	.0724
Dashed circle	63.704	13.542	4.7041
Striped circle	32.182	14.780	2.1774
Filled polygon	1.256	11.324	.1109
Filled masked polygon	44.524	32.264	1.3799
Filled textured polygon	34.866	33.408	1.0436
Polygon	3.924	10.270	.3820
Dashed polygon	25.576	10.328	2.4763
Striped polygon	50.972	11.184	4.5575
Text	23.790	60.460	.3934
Copyarea	3.542	37.498	.0944
Average time	47.734	23.521	2.029

Table 5.2: Xlib vs OpenGL performance: Avg. of 5, 100,000 primitives (Machine 1)

OpenGL implementation on machine 1 and 3.610 times faster on machine 2 (see Table 5.5).

The OpenGL implementation of Unicon's 2D graphics facilities perform quite well against the Xlib implementation considering the amount of memory allocation required of the display list architecture. The Xlib implementation has the definitive upper hand on rendering filled and unfilled primitives, text, and copyareas. However, the OpenGL implementation has the upper hand when rendering all 26 primitives tested.

	Xlib (sec)	OpenGL (sec)	Xlib-to-OpenGL ratio
Filled rectangle	0.984	1.854	.5307
Filled masked rectangle	6.660	3.490	1.9083
Filled textured rectangle	7.814	3.596	2.1729
Rectangle	5.562	1.892	2.9397
Dashed rectangle	8.362	2.002	4.1768
Striped rectangle	13.476	2.006	6.7178
Filled arc	.956	4.262	0.2243
Filled masked arc	9.974	5.896	1.6916
Filled textured arc	9.934	6.350	1.5644
Arc	3.692	4.358	.8471
Dashed arc	22.346	4.142	5.3949
Striped arc	23.172	4.516	5.1310
Filled circle	0.932	4.220	.2208
Filled masked circle	9.404	5.840	1.6102
Filled textured circle	9.498	6.546	1.4509
Circle	1.064	4.330	.2457
Dashed circle	18.212	4.134	4.4054
Striped circle	16.896	4.700	3.5948
Filled polygon	1.046	1.968	.5315
Filled masked polygon	7.734	3.924	1.9709
Filled textured polygon	8.554	3.800	2.2510
Polygon	1.196	1.902	.6288
Dashed polygon	4.330	1.886	2.2958
Striped polygon	9.200	1.936	4.7520
Text	10.994	7.844	1.4015
Copyarea	0.544	27.994	.0194
Average time	8.174	4.823	1.695

Table 5.3: Xlib vs OpenGL performance: Avg. of 5, 100,000 primitives (Machine 2)

5.2.2 ANIMATION

The program **animation** is based off of **speedtest**. However, instead of exiting after rendering all primitives, each primitive is animated by using display list record modifiable fields. The FPS is calculated by using Unicon's **&time** keyword and keeping a rolling average of the last 50 calculated FPS values. The goal of this test is to discover how much faster a dedicated Nvidia GPU is than an integrated GPU.

	Avg. runtime (sec) Xlib	Avg. runtime (sec) OpenGL	Xlib-to-OpenGL ratio
Machine 1	47.734	23.521	2.029
Machine 2	8.174	4.823	1.695

Table 5.4: Xlib vs OpenGL avg. performance: Avg. of 5, 100,000 primitives

	Avg. runtime (sec) Xlib	Avg. runtime (sec) OpenGL	Xlib-to-OpenGL ratio
Machine 1	4.738	20.797	0.228
Machine 2	1.677	6.062	0.277

Table 5.5: Xlib vs OpenGL avg. performance: Avg. of 5, 100,000 primitives (no **linestyle/fillstyle**)

The two fastest and slowest rendered primitives were picked from the **speedtest** results to be used to animating. These four primitives were run on each machine, with the number of primitives being tweaked to find a relatively steady FPS of around 30 (results in Figures 5.6 and 5.7).

	# of primitives	Display list size	FPS	Primitives per second
Filled rectangle	13,000	26,013	29	377,000
Filled polygon	3,500	14,013	29	101,500
Text	600	1,566	29	17,400
Copyarea	5,500	5,513	29	159,500

Table 5.6: OpenGL animation performance (Machine 1)

	# of primitives	Display list size	FPS	Primitives per second
Filled rectangle	41,500	83,013	31	1,286,500
Filled polygon	11,500	46,013	30	345,000
Text	1,400	3,638	31	43,400
Copyarea	18,000	18,013	29	522,000

Table 5.7: OpenGL animation performance (Machine 2)

At about 30 FPS, machine 2 can render 3.413 time more rectangles, 3.399 times more star polygons, 2.494 time more text strings (**"This is a string!"**) of length 17, and 3.273 times more copyareas of the Unicon logo than machine 1.

The OpenGL implementation of Unicon's 2D graphics facilities performs well both qualitatively and quantitatively. This new implementation is robust enough to handle most standard Icon and Unicon 2D graphics programs similarly to legacy Xlib implementation. In order for the OpenGL implementation to replace the Xlib implementation, more fixes and improvements are needed. Unexpected rendering artifacts aside, there could be an issue with display list sizes growing too large to be practical for complex GUIs, such as UI, without any display list optimization. See Appendix B.1.3 for display list optimizations available for the end user.

Chapter 6: Conclusion

The evaluation in the preceding chapter indicates that overall, this thesis project was a success. The OpenGL implementation of 2D facilities is both feature-complete and competitive performance-wise to the legacy 2D facilities. The qualitative evaluation of UI and IVIB indicate that the display list architecture scales well enough to handle complex GUI applications. The integrated 2D/3D graphics mode has the potential to replace legacy subwindow-based applications. The GUI class library needs to be modified to support double-buffering along with its software-based buffering with Unicon hidden windows.

6.1 Release

The OpenGL implementation of the Unicon 2D graphics facilities and the integrated 2D/3D graphics mode are not production grade, but are fast and accurate enough to run most existing Unicon graphics programs. The OpenGL implementation should not yet replace the Xlib implementation, but should be released alongside it as a public beta.

The purpose of the public beta is to expose the OpenGL to various machines which may or may not produce unexpected behavior. The public beta will also expose implementation bugs that a single test would not be able to find. Through exposure to more machine types and Unicon users, the OpenGL implementation can be made more robust. The decision to replace the Xlib implementation will come once the OpenGL implementation is proved proven to be as stable and reliable as the Xlib implementation.

6.2 Future Work

There are no optimizations for the display list. The most significant consequence of a display list architecture is that its performance scales linearly with its size. It would be possible to create algorithms for the display list to remove obscured and redundant

items in addition to combining multiple same items into one. However, optimizations come at the cost of surprises for the end user. The goal of the display list architecture was to provide graphics facilites that are intuitive and easy to use. If implemented, the user should be given of option of deciding the level of display list optimization. Further perforamance gains could likely be gained from moving away from OpenGL's fixed pipeline and use buffer objects. This would remove the GPU and CPU synchronization bottleneck if vertex data is managed properly.

There are also improvements that can be made to elevate this implementation's graphical accuracy and feature-completeness. One area that needs improvement is OpenGL's handling of thick lines. Not only does OpenGL not guarantee a certain line thickness, but also the quality of thick lines that OpenGL produces is questionable and does not handle capping or joining. A piece of future work is to implement both a thick line rendering algorithm as well as capping and joining algorithms. Another avenue of future work are changes to the GUI class library to better support the OpenGL implementation in order to further the integrated 2D/3D mode.

However, the arguably most important piece of future work is to implement windowing system interface code for Windows and macOS in order to utilize OpenGL's cross-platform nature. A significant factor for motivating the port of the 2D facilities to OpenGL was to provide graphics facilities for the three major operating systems. There is also the option of porting to OpenGL ES to extend the reach of Unicon graphics to Android.

References

[1] Documentation for gosu/gosu. https://www.rubydoc.info/github/gosu/gosu, April. Date Accessed: 04-15-2020.

[2] Gosu. libgosu.org. Date Accessed: 04-15-2020.

[3] Languages using tk. https://tkdocs.com/resources/languages.html. Date Accessed: 04-15-2020.

[4] Opengl es. https://developer.android.com/guide/topics/graphics/opengl. Date Accessed: 04-15-2020.

[5] Opengl es overview. https://www.khronos.org/opengles/. Date Accessed: 04-15-2020.

[6] raylib. raylib.com. Date Accessed: 04-15-2020.

[7] Simple and fast multimedia library. https://www.sfml-dev.org/index.php. Date Accessed: 04-15-2020.

[8] tkinter—python interface to tcl/tk. docs.python.org/3/library/tkinter.html. Date Accessed: 04-15-2020.

[9] Rory Daulton. How do i efficiently determine if a polygon is convex, non-convex, or complex? Stack Overflow. https://stackoverflow.com/questions/471962/how-do-i-efficiently-determine-if-a-polygon-is-convex-non-convex-or-complex/45372025#45372025, Date Accessed: 04-15-2020.

[10] Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend. *Graphics Programming in Icon*. Peer-to-Peer Communications, Inc., USA, 1998.

[11] Clinton Jeffery. Unicon: an extended dialect of icon. unicon.org/. Date Accessed: 04-15-2020.

[12] Clinton Jeffery, Naomi Martinez, and Jafar Al Gharaibeh. The implementation of graphics in unicon version 12. Technical report, University of Idaho, Department of Comupter Science, Moscow, ID, 2014.

[13] Renate Kempf and Jed Hartman. Opengl on silicon graphics systems. `http://www-f9.ijs.si/~matevz/docs/007-2392-003/sgi_html/index.html`, 1996-2005. Date Accessed: 04-15-2020.

[14] David Rosenthal. Inter-client communication conventions manual version 1.1: Mit x consortium standard. `http://lesstif.sourceforge.net/doc/super-ux/g1ae04e/chap3-1.html#Preface%20to%20Version%202.0`, 1988-1994. Date Accessed: 04-15-2020.

[15] John W. Shipman. Tkinter 8.5 reference: a gui for python. `https://web.archive.org/web/20190524140835/https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html`, December. Date Accessed: 04-15-2020.

[16] tcltk. The tk widget toolkit for tcl. `https://github.com/tcltk/tk`, 2019. Date Accessed: 04-15-2020.

[17] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide (2nd Ed.): The Official Guide to Learning OpenGL Version 1.1.* Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[18] John Zelle. *Python Programming: An Introduction to Computer Science.* Franklin, Beedle & Associates Inc., 2154 NE Broadway, Swuite 100, Portland, Oregon, 3rd edition, 2017. Date Accessed: 04-15-2020.

Appendix A: Developer Reference

For any Unicon developer looking to work with the OpenGL implementation of Unicon's 2D and integrated 2D/3D graphics facilities, read section 4 to get an overview of how the system was designed.

The macros specific to the OpenGL implementation are located in **src/h/opengl.h**. The bulk of the 2D OpenGL implementation is located in **src/runtime/ropengl2d.ri**, with the windowing code residing in **src/runtime/rxwin.ri** and **src/runtime/rxrsc.ri**. For those wishing to work on windowing code, go to **src/runtime/rmswin.ri** for Windows and **src/runtime/rmac.ri** and **rmacrsc.ri** for macOS. For those looking to work on the integrated 2D/3D facilities, look in **src/runtime/ropengl.ri**, **src/runtime/fwindow.r**, and **src/runtime/rwindow.r**. As implied by its namesake, the integrated 2D/3D facilities are integrated into Unicon's graphics facilities and are therefore harder to pin down to a specific file.

Unicon's OpenGL implementation of the 2D and integrated 2D/3D facilities are not robust enough to replace the legacy Xlib implementation. Thus, the macro **GraphicsGL** is used to conditionally define 2D and integrated 2D/3D OpenGL implementation code via **#ifdef**s.

A.1 Internal OpenGL 2D API

The Unicon 2D graphics facilities has an internal API that can be found in Unicon Technical Report 5b. OpenGL's implementation mimics the 2D facilities internal API by prepending **"gl_"** to each function. This prefix will remain until the implementation is deemed robust enough to replace the legacy implementation. A subset of the windowing functions are platform-specific (GLX for Linux, WGL for Windows, and CLG for macOS) due to GLUT's lack of complete windowing functionality.

A.1.1 Core API

The core API is untouched with the except of an occasional added parameter and the addition of functions **gl_drawcircles()**, **gl_fillcircls()**, **gl_arcs()**, **gl_rectangles()**, and **gl_colors()**.

Rendering

int gl_blimage(wbp w, int x, int y, int width, int height, int ch, unsigned char *s, word len)

int gl_readimage(wbp w, char *filename, int x, int y, int *status)

int gl_strimage(wbp w, int x, int y, int width, int height, struct palentry *e, unsigned char *s, word len, int on_icon)

int gl_drawstrng(wbp w, int x, int y, char *s, int slen)

int gl_xdis(wbp w, char *s, int s_len)

int gl_copyArea(wbp w1, wbp w2, int x, int y, int width, int height, int x2, int y2)

int gl_eraseArea(wbp w, int x, int y, int width, int height)

int gl_arcs(wbp w, XArc *arcs, int n, int circle, int fill)

int gl_fillarcs(wbp w, XArc *arcs, int n)

int gl_drawarcs(wbp w, XArc *arcs, int n)

int gl_fillcircles(wbp w, XArc *arcs, int n)

int gl_drawcircles(wbp w, XArc *arcs, int n)

int gl_rectangles(wbp w, XRectangle *recs, int n, int fill)

int gl_fillrectangles(wbp w, XRectangle *recs, int n)

int gl_drawrectangles(wbp w, XRectangle *recs, int n)

int gl_drawlines(wbp w, XPoint *points, int n)

int gl_drawpoints(wbp w, XPoint *points, int n)

int gl_drawsegments(wbp w, XSegment *segs, int n)

int gl_fillpolygon(wbp w, XPoint *pts, int n)

int gl_dumpimage(wbp w, char *filename, unsigned int x, unsigned int y, unsigned int width, unsigned int height)

int gl_getimstr(wbp w, int x, int y, int width, int height, struct palentry *ptbl, unsigned char *data)

int gl_getimstr24(wbp w, int xx, int yy, int width, int height, unsigned char *data)

int gl_getpixel_init(wbp w, struct imgmem *imem)

int gl_getpixel_term(wbp w, struct imgmem *imem)

int gl_getpixel(wbp w, int x, int y, long *rv, char *s, struct imgmem *imem)

char *gl_loadimage(wbp w, char *filename, unsigned int *height, unsigned int *width,
int atorigin, int *is_pixmap)

Context Attributes

void gl_getfg(wbp w, char *s)

void gl_getbg(wbp w, char *s)

void gl_getdrawop(wbp w, char *s)

void gl_getlinestyle(wbp w, char *s)

void gl_getfntnam(wbp w, char *s)

char *gl_get_mutable_name(wbp w, int mute_index)

int gl_set_mutable(wbp w, int mute_index, char *s)

void gl_free_mutable(wbp w, int mute_index)

int gl_mutable_color(wbp w, dptr argv, int warg, int *rv)

int gl_color(wbp w, int intcode, int mindex, char *s)

int gl_setbgrgb(wbp w, int r, int g, int b)

int gl_setfgrgb(wbp w, int r, int g, int b)

int gl_isetbg(wbp w, int mindex)

int gl_isetfg(wbp w, int mindex)

int gl_setbg(wbp w, char *s)

int gl_setfg(wbp w, char *s)

int gl_setdrawop(wbp w, char *s)

int gl_setleading(wbp w, int i)

int gl_setlinestyle(wbp w, char *s)

int gl_setlinewidth(wbp w, LONG lwidth)

int gl_SetPattern(wbp w, char *name, int len)

int gl_setfillstyle(wbp w, char *s)

int gl_setfont(wbp w, char **s)

wfp gl_alc_font(wbp w, char **s, int len)

int gl_setgamma(wbp w, double gamma)

int gl_setclip(wbp w)

int gl_unsetclip(wbp w)

int gl_setdx(wbp w)

int gl_setdy(wbp w)

int gl_toggle_fgbg(wbp w)

Windowing

int gl_allowresize(wbp w, int on)

char gl_child_window_stuff(wbp w, wbp wp, int child_window)

wcp gl_clone_context(wbp w)

int gl_rebind(wbp w, wbp w2)

int gl_resizePixmap(wbp w, int width, int height)

int gl_setcursor(wbp w, int on)

int gl_wputc(int c, wbp w)

A.1.2 Platform-specific API

Until the OpenGL implementation replaces the Xlib implementation, these platform-specific functions call their non-prefixed counterparts. Some functions (**gl_alc*()** and **gl_free*()**) execute OpenGL-specific code as well.

Windowing

wcp gl_alc_context(wbp w)

wdp gl_alc_display(char *s)

wsp gl_alc_winstate()

void gl_free_context(wcp wc)

void gl_free_display(wdp wd)

int gl_free_window(wsp ws)

void gl_freecolor(wbp w, char *s)

int gl_do_config(wbp w, int status)

void gl_getcanvas(wbp w, char *s)

int gl_getdefault(wbp w, char *prog, char *opt, char *answer)

void gl_getdisplay(wbp w, char *s)

void gl_geticonic(wbp w, char *s)

int gl_geticonpos(wbp w, char *s)

void gl_getpointername(wbp w, char *s)

int gl_getpos(wbp w)

int gl_getvisual(wbp w, char *s)

int gl_nativecolor(wbp w, char *s, long *r, long *g, long *b)

int gl_lowerWindow(wbp w)

int gl_raiseWindow(wbp w)

int gl_setcanvas(wbp w, char *s)

int gl_setdisplay(wbp w, char *s)

int gl_seticonicstate(wbp w, char *s)

int gl_seticonimage(wbp w, dptr dp)

int gl_seticonlabel(wbp w, char *s)

int gl_seticonpos(wbp w, char *s)

int gl_setimage(wbp w, char *s)

int gl_setpointer(wbp w, char *s)

int gl_setwidth(wbp w, SHORT new_width)

int gl_setheight(wbp w, SHORT new_height)

int gl_setgeometry(wbp w, char *s)

int gl_setwindowlabel(wbp w, char *s)

int gl_query_pointer(wbp w, XPoint *xp)

int gl_query_rootpointer(XPoint *xp)

int gl_walert(wbp w, int volume)

void gl_warpPointer(wbp w, int x, int y)

int gl_wclose(wbp w)

void gl_wflush(wbp w)

void gl_wflushall()

int gl_wgetq(wbp w, dptr res, int t)

FILE *gl_wopen(char *name, struct b_list *lp, dptr attrs, int n, int *err_index, int is_3d)

int gl_wmap(wbp w)

void gl_wsync(wbp w)

A.2 Display List Objects

All display list objects are detailed, from their Unicon types and which Unicon procedures and internal C functions they are created from. At this time, all display list objects are records with semantically enforced private fields which are prepended with double underscores (__). The runtime system will try to convert public fields to their defined types. See macros **GetStr()**, **GetInt()**, and **GetDouble()** in **src/h/opengl.h** for details. Private fields should not be modified to values which are not the specified types. Doing so will result in undefined behavior.

Image Primitives

---

gl2d_blimage: record — **DrawImage(): gl_blimage()**

name: "BilevelImage", **code:** GL2D_BLIMAGE,

x: Real, **y:** Real, **width:** Int, **height:** Int, **__s:** String, **__ch:** Int, **__texid:** Int, **__index:** Int

---

gl2d_readimage: record — **ReadImage(): gl_readimage()**

name: "ReadImage", **code:** GL2D_BLIMAGE,

x: Real, **y:** Real, **width:** Int, **height:** Int, **__is_pixmap:** Int, **__pixmap:** String, **__texid:** Int, **__index:** Int

---

gl2d_strimage: record — **DrawImage()/ReadImage(): gl_strimage()**

name: "StringImage", **code:** GL2D_STRIMAGE,

x: Real, **y:** Real, **width:** Int, **height:** Int, **__pixmap:** String, **__texid:** Int, **__index:** Int

---

gl2d_drawstring: record — **DrawString(): gl_drawstrng()**

name: "DrawString", **code:** GL2D_DRAWSTRING,

x: Real, **y:** Real, **s:** Int

---

gl2d_wwrite: record — **WWrite*(): gl_xdis()**

name: "WWrite", **code:** GL2D_WWRITE,

x: Real, **y:** Real, **s:** Int

---

gl2d_copyarea: record **CopyArea(): gl_copyArea()**

name: "CopyArea", **code:** GL2D_COPYAREA,

__x1: Real, **__y1:** Real, **__width:** Int, **__height:** Int, **x:** Real, **y:** Real, **__texid:** Int, **__index:** Int

---

gl2d_erasearea: record **EraseArea(): gl_eraseArea()**

name: "EraseArea", **code:** GL2D_ERASEAREA,

x: Real, **y:** Real, **width:** Real, **height:** Real

Geometric Primitives

---

gl2d_fillpolygon: record **FillPolygon(): gl_fillpolygon()**

name: "FillPolygon", **code:** GL2D_FILLPOLYGON,

__v: Realarray, **__coords:** Realarray, **__type:** Int

---

gl2d_drawpolygon: record **DrawPolygon(): gl_drawlines()**

name: "DrawPolygon", **code:** GL2D_DRAWPOLYGON,

__v: Realarray, **__coords:** Realarray, **__vseg:** Realarray

---

gl2d_drawline: record **DrawCurve()/DrawLine(): gl_drawlines()**

name: "DrawLine", **code:** GL2D_DRAWLINE,

__v: Realarray, **__coords:** Realarray, **__vseg:** Realarray

---

gl2d_drawsegment: record **DrawSegment(): gl_drawsegments()**

name: "DrawSegment", **code:** GL2D_DRAWSEGMENT,

__v: Realarray, **__coords:** Realarray, **__vseg:** Realarray

---

gl2d_drawpoint: record **DrawPoint(): gl_drawpoints()**

name: "DrawPoint", **code:** GL2D_DRAWPOINT,

__v: Realarray, **__coords:** Realarray

---

gl2d_drawcircle: record **DrawCircle(): gl_arcs()**

name: "DrawCircle", **code:** GL2D_DRAWCIRCLE,

__v: Realarray, **x:** Real, **y:** Real, **r:** Real, **theta:** Real, **alpha:** Real,

__x: Real, **__y:** Real, **__r:** Real, **__theta:** Real, **__alpha:** Real, **__vseg:** Realarray

---

gl2d_fillcircle: record **FillCircle(): gl_arcs()**

name: "FillCircle", **code:** GL2D_FILLCIRCLE,

__v: Realarray, **x:** Real, **y:** Real, **r:** Real, **theta:** Real, **alpha:** Real,

__x: Real, **__y:** Real, **__r:** Real, **__theta:** Real, **__alpha:** Real

---

gl2d_drawarc: record **DrawArc(): gl_arcs()**

name: "DrawArc", **code:** GL2D_DRAWARC,

__v: Realarray, **x:** Real, **y:** Real, **width:** Real, **height:** Real, **theta:** Real,

alpha: Real, **__x:** Real, **__y:** Real, **__width:** Real, **__height:** Real, **__theta:** Real,

__alpha: Real, **__vseg:** Realarray

---

gl2d_fillarc: record **FillArc(): gl_arcs()**

name: "FillArc", **code:** GL2D_FILLARC,

__v: Realarray, **x:** Real, **y:** Real, **width:** Real, **height:** Real, **theta:** Real,

alpha: Real, **__x:** Real, **__y:** Real, **__width:** Real, **__height:** Real, **__theta:** Real,

__alpha: Real

---

gl2d_drawrectangle: record **DrawRectangle(): gl_rectangles() name: "DrawRectangle"**, **code:** GL2D_DRAWRECTANGLE,

__v: Realarray, **x:** Real, **y:** Real, **width:** Real, **height:** Real **__x:** Real,

__y: Real, **__width:** Real, **__height:** Real, **__vseg:** Realarray

---

gl2d_fillrectangle: record **FillRectangle(): gl_rectangles() name: "FillRectangle"**, **code:** GL2D_FILLRECTANGLE,

__v: Realarray, **x:** Real, **y:** Real, **width:** Real, **height:** Real **__x:** Real,

__y: Real, **__width:** Real, **__height:** Real

---

gl2d_fg: record **WAttrib()/Fg(): gl_color()**

name: "Fg", **code:** GL2D_FG,

r: Int, **g:** Int, **b:** Int, **a:** Int

---

gl2d_bg: record **WAttrib()/Bg(): gl_color()**

name: "Bg", **code:** GL2D_BG,

r: Int, **g:** Int, **b:** Int, **a:** Int

---

gl2d_reverse: record **WAttrib(): gl_togglefgbg()**

name: "Reverse", **code:** GL2D_REVERSE

---

gl2d_gamma: record **WAttrib(): gl_setgamma()**

name: "Gamma", **code:** GL2D_GAMMA,

val: Real

---

gl2d_drawop: record **WAttrib(): gl_setdrawop()**

name: "Drawop", **code:** GL2D_DRAWOP,

s: String

---

gl2d_font: record **WAttrib()/Font(): gl_setfont()**

name: "Font", **code:** GL2D_FONT,

s: String

---

gl2d_leading: record **WAttrib(): gl_setleading()**

name: "Leading", **code:** GL2D_LEADING,

val: Int

---

gl2d_linewidth: record **WAttrib(): gl_setlinewidth()**

name: "Linewidth", **code:** GL2D_LINEWIDTH,

val: Int

---

gl2d_linestyle: record **WAttrib(): gl_setlinestyle()**

name: "Linestyle", **code:** GL2D_LINESTYLE,

s: String

---

gl2d_fillstyle: record **WAttrib(): gl_setfillstyle()**

name: "Fillstyle", **code:** GL2D_FILLSTYLE,

s: String

---

gl2d_pattern: record **WAttrib(): gl_SetPattern()**

name: "Pattern", **code:** GL2D_PATTERN,

s: String, **__s:** String, **__width:** Int, **__height:** Int, **__texid:** Int, **__index:** Int

---

gl2d_clip: record **WAttrib(): gl_setclip()**

name: "Clip", **code:** GL2D_CLIP,

x: Int, **y:** Int, **width:** Int, **height:** Int

---

gl2d_dx: record **WAttrib(): gl_setdx()**

name: "Dx", **code:** GL2D_DX,

val: Int, **__val:** Int

---

gl2d_dy: record **WAttrib(): gl_setdy()**

name: "Dy", **code:** GL2D_DY,

val: Int, **__val:** Int

Appendix B: User Guide

Here is a Unicon user guide for the new implementation of the 2D graphics facilities. This implementation was designed to be backwards compatible, so please use all the Unicon graphics knowledge at your disposal. To use the 2D OpenGL implementation on Linux machines, set the environment variable **UNICONGL2D** and use mode **"g"** for a 2D-only window and mode **"gl"** for an integrated 2D/3D window.

B.1 2D Features

The display list offers new capabilities to the 2D facilities, including transparency and a display list.

B.1.1 Transparency

There are three different ways of using transparency. The first is with the new context attribute **alpha** which can have a value ranging from 0 (fully transparent) to 1 (fully opaque), inclusive. The default value of **alpha** is 1.0. Any object rendered with an alpha of 0 will not be visible. The second is by specifying it explicitly in a color specification, i.e. **"fg=#FF000080"**, **"fg=65535,0,0,32768"**, or **"fg=translucent red"**. The color phrase modifiers from the 3D facilities can be found in Table 3 of Section 2.8 of Unicon Technical Report 9b. The third way is by modifying the alpha field, **a**, of a **Fg** or **Bg** display list entry (read the following sections for more information).

When a color specification does not context an explicit alpha value, then **alpha** is used. When a color specification contains an explicit alpha value, then the context attribute is ignored and the specified alpha is used.

To ensure proper transparency blending, any transparent objects should be drawn in the correct order, i.e. objects that appear farther away should be drawn first. Once drawn in the correct order, the display list will preserve the order, unless the ordering list is modified by the user.

B.1.2 Animation with a Display List

The addition of a display list offers the ability for intuitive animation. Instead of redrawing everything with new positions, it is possible to grab references to desired primitives and change certain attributes. See Appendix B.3 for details on which fields of display list records are modifiable.

All objects on the display lists are records, each possessing the field **name** which contains the string literal identifying its record type (see Appendix B.3). The modification does not take effect until the display list is redrawn with Unicon procedure **Refresh()**. The Unicon procedure **WindowContents()** returns a reference to the display list.

The reference to a primitive can be obtained by grabbing the last element of the display list after returning from a successful call to a Unicon graphics procedure that creates a display list record.

```
DrawRectangle(0,0,100,100)
drawrect := WindowContents()[-1]
```

Listing B.1: Obtaining the reference to a display list record

All graphical primitives except **FillPolygon**, **DrawPolygon**, **DrawLine**, **DrawSegment**, and **DrawPoint** have **x** and **y** fields to be used for modifying the position of a particular primitives.

```
drawrect.x +:= 5
drawrect.y +:= 5
Refresh()
```

Listing B.2: Changing the position of a primitive using **x** and **y** fields

For graphical primitives that do not possess positional fields, the translational coordinates **dx** and **dy** can be used instead. Display list records **Dx** and **Dy** affect the translational coordinates of all subsequent graphical primitives until another translational display list record of the same type is encountered. Thus, **Dx** and **Dy** can be used to translate groups of primitives. This is more efficient than modifying each **x** and **y** for each graphical primitive that possesses those fields.

Context attribute display list records are created and appended to the display list by **WAttrib()** in order of the argument list given. Only assignments create display list records. However, all records are created before the first suspension of **WAttrib()**.

```
dl := WindowContents()
WAttrib("dx=0","dy=0")
dx := dl[-2]
dy := dl[-1]
DrawPolygon(0,0,0,100,100,100,100,0)
DrawLine(0,0,100,100)
DrawLine(0,100,100,0)

dx.val +:= 5
dy.val +:= 5
Refresh()
```

Listing B.3: Changing the position of primitives using **Dx** and **Dy**

Another technique is to build a list of references to display list records that need to be modified in a render loop. A case statement can be used on the **name** field of the records to differentiate between the types of records. Listing B.4 for the Unicon source code of a program that uses a simple animation loop. The result is a color-shifting, filled circle that bounces off the walls of the window. For another example of the new animation semantics, see Listing C.2 in Appendix C.

```
link graphics
\$include "keysyms.icn"

procedure main()
   width := height := 400
   L := []
   &window := open("simple animation", "g", "size="||width||","||height)
   dl := WindowContents()

   WAttrib("fg=red")
   put(L, dl[-1])
   FillCircle(10,200,10)
   put(L, dl[-1])

   dirx := diry := 1
   speed := .1
   repeat {
      if *Pending() > 0 then {
         case Event() of {
            "q": exit(0)
            }
         }
      else every obj := !L do {
         case obj.name of {
            "FillCircle": {
               if (obj.x + dirx*(obj.r + speed) > width) then
                  dirx := -1
               else if (obj.x + dirx*(obj.r + speed) < 0) then
```

```
                dirx := 1
             if (obj.y + diry*(obj.r + speed) > height) then
                diry := -1
             else if (obj.y + diry*(obj.r + speed) < 0) then
                diry := 1
             obj.x +:= speed*dirx
             obj.y +:= speed*diry
             }
          "Fg": {
             tmp := obj.r
             obj.r := obj.g
             obj.g := obj.b
             obj.b := tmp
             }
          }
       }
    WDelay(1)
    Refresh()
    }
end
```

Listing B.4: Simple animation loop program

B.1.3 Optimizing Performance

Now that the 2D facilities utilize a display list, it is in the best interest of everyone involved to manage its size. Consequently, Unicon procedures **CopyArea()** and **EraseArea()** can be used to manage display list size if called with either default arguments or positional and dimensional arguments that cover the entire window, i.e. **CopyArea()**, **CopyArea(w)**, **CopyArea(0,0,win_width,win_height,0,0)**, etc.

Calling **CopyArea()** with positional and dimensional arguments to cover the entire destination window will delete the destination window's display list and add a CopyArea display list record. This is useful when all entries before a specific point on one window no longer needs to be animated.

Calling **EraseArea()** called with positional and dimensional arguments to cover the entire window will cause the display list to be deleted. If subsequent drawing will fully obscure previous display list records, then call **EraseArea(w)** to remove those records.

B.2 Integrated 2D/3D Features

The integrated 2D/3D facilities are offered as an optional extension to Unicon's 3D mode, **"gl"**. The context attribute **rendermode** is used to switch between 2D

(**"rendermode=2d"**) and 3D (**"rendermode=3d"**) render modes. **rendermode** defaults to 3D rendering unless otherwise specified in the attribute list given to **open()**. **rendermode** defaults to the value of the target context to **Clone()** unless otherwise specified in its given attribute list. One can switch rendermodes using only one context as well as using two or more contexts and designating a specific **rendermode** to each.

```
# Draw 2D (HUD) stuff
WAttrib("rendermode=2d")
DrawPolygon(...)
...
# Draw 3D stuff
WAttrib("rendermode=3d")
DrawPolygon(...)
...
```

Listing B.5: Rendermode switching with one canvas and one context

```
w3d := open("", "gl", ...)
w2d := Clone(w3d, "rendermode=2d", ...)

# Draw 2D (HUD) stuff
DrawPolygon(w2d, ...)
...
# Draw 3D stuff
DrawPolygon(w3d, ...)
...
```

Listing B.6: Rendermode switching with one canvas and two contexts

Unicon procedures **Eye()** and **Refresh()** redraw both 2D and 3D display lists (if present). It is recommended to use double-buffering, **"buffer=on"**, to reduce screen flickering. The semantics for the 2D and 3D facilities remain the same, but now exist in one graphics mode. Remember to use the proper **rendermode** or else a runtime error will likely occur.

B.3 Modifiable Display List Record Fields

The display list records with their modifiable fields are shown as Unicon record declarations. Every display list record has the field **name**, containing the string literal which identifies it. **name** should not be modified. It is shown here as useful information rather than a modifiable field. If no fields other than **name** are shown, it means that no modifiable fields are available. It is possible to query the fields of these records at runtime,

but modifying any fields other than the ones shown in this section will result in undefined behavior.

Primitives

---

gl2d_blimage: record **DrawImage()**

name: "BilevelImage",

x: Real, **y:** Real, **width:** Int, **height:** Int,

---

gl2d_readimage: record **ReadImage()**

name: "ReadImage",

x: Real, **y:** Real, **width:** Int, **height:** Int,

---

gl2d_strimage: record **DrawImage()/ReadImage()**

name: "StringImage",

x: Real, **y:** Real, **width:** Int, **height:** Int,

---

gl2d_drawstring: record **DrawString()**

name: "DrawString",

x: Real, **y:** Real, **s:** Int

---

gl2d_wwrite: record **WWrite*()**

name: "WWrite",

x: Real, **y:** Real, **s:** Int

---

gl2d_copyarea: record **CopyArea()**

name: "CopyArea",

x: Real, **y:** Real,

x and **y** define the coordinates of the destination window.

---

gl2d_erasearea: record **EraseArea()**

name: "EraseArea",

x: Real, **y:** Real, **width:** Real, **height:** Real

---

gl2d_fillpolygon: record	**FillPolygon()**
name: "FillPolygon"	
gl2d_drawpolygon: record	**DrawPolygon()**
name: "DrawPolygon"	
gl2d_drawline: record	**DrawCurve()/DrawLine()**
name: "DrawLine"	
gl2d_drawsegment: record	**DrawSegment()**
name: "DrawSegment"	
gl2d_drawpoint: record	**DrawPoint()**
name: "DrawPoint"	
gl2d_drawcircle: record	**DrawCircle()**
name: "DrawCircle", **x:** Real, **y:** Real, **r:** Real, **theta:** Real, **alpha:** Real	
gl2d_fillcircle: record	**FillCircle()**
name: "FillCircle", **x:** Real, **y:** Real, **r:** Real, **theta:** Real, **alpha:** Real	
gl2d_drawarc: record	**DrawArc()**
name: "DrawArc", **x:** Real, **y:** Real, **width:** Real, **height:** Real, **theta:** Real, **alpha:** Real	
gl2d_fillarc: record	**FillArc()**
name: "FillArc", **x:** Real, **y:** Real, **width:** Real, **height:** Real, **theta:** Real, **alpha:** Real	
gl2d_drawrectangle: record	**DrawRectangle()**
name: "DrawRectangle", **x:** Real, **y:** Real, **width:** Real, **height:** Real	
gl2d_fillrectangle: record	**FillRectangle()**

name: "FillRectangle",

x: Real, **y:** Real, **width:** Real, **height:** Real

Context Attributes

---

gl2d_fg: record **WAttrib()/Fg()**

name: "Fg",

r: Int, **g:** Int, **b:** Int, **a:** Int

---

gl2d_bg: record **WAttrib()/Bg()**

name: "Bg",

r: Int, **g:** Int, **b:** Int, **a:** Int

---

gl2d_reverse: record **WAttrib()**

name: "Reverse"

---

gl2d_gamma: record **WAttrib()**

name: "Gamma",

val: Real

---

gl2d_drawop: record **WAttrib()**

name: "Drawop",

s: String

---

gl2d_font: record **WAttrib()/Font()**

name: "Font",

s: String

---

gl2d_leading: record **WAttrib()**

name: "Leading",

val: Int

---

gl2d_linewidth: record **WAttrib()**

name: "Linewidth",

val: Int

---

gl2d_linestyle: record **WAttrib()**

name: "Linestyle",

s: String

---

gl2d_fillstyle: record **WAttrib()**

name: "Fillstyle",

s: String

---

gl2d_pattern: record **WAttrib()**

name: "Pattern",

s: String

---

gl2d_clip: record **WAttrib()**

name: "Clip",

x: Int, **y:** Int, **width:** Int, **height:** Int

---

gl2d_dx: record **WAttrib()**

name: "Dx",

val: Int

---

gl2d_dy: record **WAttrib()**

name: "Dy",

val: Int

Appendix C: Unicon Test Programs

C.1 speedtest.icn

```
link graphics
\$include "keysyms.icn"

global patterns, text, colors, alphas, mode

procedure main(argv)
   &random := 0
   width := height := 800
   usage := "usage: speedtest primitive N [style]\n"||
      "\tprimitive: 1 - filled rectangles\n"||
      "\t           2 - unfilled rectangles\n"||
      "\t           3 - filled arcs\n"||
      "\t           4 - unfilled arcs\n"||
      "\t           5 - filled circles\n"||
      "\t           6 - unfilled circles\n"||
      "\t           7 - filled star polygons\n"||
      "\t           8 - unfilled star polygons\n"||
      "\t           9 - text strings\n"||
      "\t          10 - copyareas\n"||
      "\tN: A positive integer specifying the number of randomly positioned\n"||
      "\t   primitives drawn\n"
      "\tstyle: \"masked\", \"textured\", \"dashed\", or \"striped\"\n"

   patterns := ["black", "checkers", "darkgray", "diagonal", "grains", "gray",
                "grid", "horizontal", "lightgray", "scales", "trellis",
                "vertical", "verydark", "verylight", "waves", "white"]
   colors := ["black", "blue", "brown", "cyan", "gray", "green", "magenta",
              "orange", "pink", "purple", "red", "violet", "white", "yellow"]
   alphas := ["transparent", "subtransparent", "translucent", "subtranslucent",
              "opaque"]
   text := "This is a string!"
   objs := []

   if 0 < *argv <= 3 then {
      n := integer(argv[2]) | stop(usage)

      attribs := ["speed test","g","size="||width||","||height,"bg=black"]
      case (argv[3]|"") of {
         "masked": {
            &window := open ! (attribs ||| ["fillstyle=masked"])
            mode := "pattern"
            }
         "textured": {
            &window := open ! (attribs ||| ["fillstyle=textured"])
            mode := "pattern"
            }
         "dashed": {
            &window := open ! (attribs ||| ["linestyle=dashed"])
            mode := ""
            }
         "striped": {
            &window := open ! (attribs ||| ["linestyle=striped"])
            mode := ""
            }
         default: {
```

```
            &window := open ! attribs
            mode := ""
            }
         }

      Bg("white")
      case argv[1] of {
         "1": rects(n,objs,1)
         "2": rects(n,objs,)
         "3": arcs(n,objs,1)
         "4": arcs(n,objs,)
         "5": circles(n,objs,1)
         "6": circles(n,objs,)
         "7": polygons(n,objs,1)
         "8": polygons(n,objs)
         "9": drawstrings(n,objs)
         "10": copyareas(n,objs)
         default:
            stop(usage)
         }
      }
   else stop(usage)
end

procedure rects(n,objs,fill)
   local DrawProc
   width := WAttrib("width")
   height := WAttrib("height")

   if \fill then DrawProc := FillRectangle
   else DrawProc := DrawRectangle

   every i := 1 to n do {
      x := ?width - 1
      y := ?height - 1
      wid := ?(width/2) + 10
      ht := ?(height/2) + 10

      if mode == "pattern" then
         WAttrib("pattern="||?patterns, "fg="||?colors)
      else
         WAttrib("fg="||?colors)
      DrawProc(x, y, wid, ht)
      }
end

procedure arcs(n,objs,fill)
   local DrawProc
   width := WAttrib("width")
   height := WAttrib("height")

   if \fill then DrawProc := FillArc
   else DrawProc := DrawArc

   every i := 1 to n do {
      x := ?width - 1
      y := ?height - 1
      wid := ?(width/2) + 10
      ht := ?(height/2) + 10

      if mode == "pattern" then
         WAttrib("pattern="||?patterns, "fg="||?colors)
      else
         WAttrib("fg="||?colors)
      DrawProc(x, y, wid, ht)
```

```
      }
end

procedure circles(n,objs,fill)
   local DrawProc
   width := WAttrib("width")
   height := WAttrib("height")

   if \fill then DrawProc := FillCircle
   else DrawProc := DrawCircle

   every i := 1 to n do {
      x := ?width - 1
      y := ?height - 1
      r := ?((width+height)/8) + 5

      if mode == "pattern" then
         WAttrib("pattern="||?patterns, "fg="||?colors)
      else
         WAttrib("fg="||?colors)

      DrawProc(x, y, r)
      }
end

procedure polygons(n,objs,fill)
   width := WAttrib("width")
   height := WAttrib("height")
   range := 100
   minpoints := 10

   if \fill then DrawProc := FillPolygon
   else DrawProc := DrawPolygon

   every i := 1 to n do {
      x := ?width - 1
      y := ?height - 1
      r := ?range + 5

      if mode == "pattern" then
         WAttrib("pattern="||?patterns, "fg="||?colors)
      else
         WAttrib("fg="||?colors)

      DrawProc(x, y, x+2*r, y, x+0.375*r, y+1.25*r, x+r, y-0.75*r,
               x+1.625*r, y+1.25*r)
      }
end

procedure drawstrings(n,objs)
   width := WAttrib("width")
   height := WAttrib("height")
   fht_range := 50
   fontlist := ["AvantGarde", "Bookman", "Charter", "Courier", "Gill Sans",
                "Helvetica", "Lucida Bright", "Lucida Sans",
                "New Century Schoolbook", "Palatino", "Rockwell", "Times"]
   attlist := ["", "bold", "italic", "bold,italic"]
   every i := 1 to n do {
      fheight := ?fht_range + 8
      x := ?(width) - 1
      y := ?(height-fheight) - 1

      WAttrib("fg="||?colors)
      Font(?fontlist||","||fheight||","||?attlist)
      DrawString(x,y,text)
```

```
      }
end

procedure copyareas(n,objs)
   width := WAttrib("width")
   height := WAttrib("height")
   im_wid := im_ht := 256
   x := y := 0

   ReadImage("unicon.jpg",x,y) | write("failed to write image")
   if n = 1 then return
   every i := 2 to n do {
      x_next := ?(width-im_wid) - 1
      y_next := ?(height-im_ht) - 1
      CopyArea(x, y, im_wid, im_ht, x_next, y_next)
      x := x_next
      y := y_next
      }
end
```

Listing C.1: Source code for **speedtest.icn**

C.2 animation.icn

```
link graphics
\$include "keysyms.icn"

global patterns, text, colors, alphas, mode
\$define M_PI 3.141519265359
\$define BINARY_RAND (-1^?2)

procedure main(argv)
   &random := 0
   width := height := 800
   usage := "usage: animation primitive N [style]\n"||
      "\tprimitive: 1 - filled rectangles\n"||
      "\t           2 - unfilled rectangles\n"||
      "\t           3 - filled arcs\n"||
      "\t           4 - unfilled arcs\n"||
      "\t           5 - filled circles\n"||
      "\t           6 - unfilled circles\n"||
      "\t           7 - filled star polygons\n"||
      "\t           8 - unfilled star polygons\n"||
      "\t           9 - text strings\n"||
      "\t          10 - copyareas\n"||
      "\tN: A positive integer specifying the number of randomly positioned\n"||
      "\t   primitives drawn\n"
      "\tstyle: \"masked\", \"textured\", \"dashed\", or \"striped\"\n"

   patterns := ["black", "checkers", "darkgray", "diagonal", "grains", "gray",
               "grid", "horizontal", "lightgray", "scales", "trellis",
               "vertical", "verydark", "verylight", "waves", "white"]
   colors := ["black", "blue", "brown", "cyan", "gray", "green", "magenta",
             "orange", "pink", "purple", "red", "violet", "white", "yellow"]
   alphas := ["transparent", "subtransparent", "translucent", "subtranslucent",
             "opaque"]
   text := "This is a string!"
   objs := []
   fps_list := []
   max_num := 50
```

```
i := 1

if 0 < *argv <= 3 then {
   n := integer(argv[2]) | stop(usage)

   attribs := ["speed test","g","size="||width||","||height,"bg=black"]
   case (argv[3]|"") of {
      "masked": {
         &window := open ! (attribs ||| ["fillstyle=masked"])
         mode := "pattern"
         }
      "textured": {
         &window := open ! (attribs ||| ["fillstyle=textured"])
         mode := "pattern"
         }
      "dashed": {
         &window := open ! (attribs ||| ["linestyle=dashed"])
         mode := ""
         }
      "striped": {
         &window := open ! (attribs ||| ["linestyle=striped"])
         mode := ""
         }
      default: {
         &window := open ! attribs
         mode := ""
         }
      }

   Bg("white")
   case argv[1] of {
      "1": rects(n,objs,1)
      "2": rects(n,objs,)
      "3": arcs(n,objs,1)
      "4": arcs(n,objs,)
      "5": circles(n,objs,1)
      "6": circles(n,objs,)
      "7": polygons(n,objs,1)
      "8": polygons(n,objs)
      "9": drawstrings(n,objs)
      "10": copyareas(n,objs)
      default:
         stop(usage)
      }
   }
else stop(usage)

# Setup fps counter
fps := 0
basex := 5
basey := 5
offsetx := 5
offsety := 5
offsetx2 := 3
offsety2 := 3

# Draw fps
WAttrib("dx=0","dy=0") # to ensure this text doesn't move
fontname := "mono"
WAttrib("font="||fontname||",78","fg=black")
fheight := WAttrib("fheight")
DrawString(basex, basey+fheight, "fps: "||string(fps))
fps_obj_shadow := WindowContents()[-1]
WAttrib("fg=white")
DrawString(basex+offsetx, basey+fheight+offsety, "fps: "||string(fps))
```

```
fps_obj := WindowContents()[-1]

# Draw display list size
WAttrib("font="||fontname||",60","fg=black")
fheight2 := WAttrib("fheight")
dl_size := *WindowContents()
DrawString(basex, basey+fheight+fheight2,"Size: "||dl_size+3)
WAttrib("fg=white")
DrawString(basex+offsetx2, basey+fheight+fheight2+offsety2,
           "Size: "||dl_size+3)

Refresh()

repeat {
   if *Pending() > 0 then {
      case ev := Event() of {
         "q": exit(0)
         }
      }
   else {
      #
      # move objects
      #
      every obj := !objs do {
         case argv[1] of {
            "1"|"2": {
               obj.x +:= BINARY_RAND*?10
               obj.y +:= BINARY_RAND*?10

               obj.width +:= BINARY_RAND*?10
               if obj.width <= 0 then obj.width := -obj.width
               obj.height +:= BINARY_RAND*?10
               if obj.height <= 0 then obj.height := -obj.height
               }
            "3"|"4": {
               obj.x +:= BINARY_RAND*?10
               obj.y +:= BINARY_RAND*?10

               obj.width +:= BINARY_RAND*?10
               if obj.width <= 0 then obj.width := -obj.width
               obj.height +:= BINARY_RAND*?10
               if obj.height <= 0 then obj.height := -obj.height

               obj.theta +:= BINARY_RAND*(?4)*M_PI/8
               obj.alpha +:= BINARY_RAND*(?4)*M_PI/8
               }
            "5"|"6": {
               obj.x +:= BINARY_RAND*?10
               obj.y +:= BINARY_RAND*?10

               obj.r +:= BINARY_RAND*?10
               if obj.r <= 0 then obj.r := -obj.r

               obj.theta +:= BINARY_RAND*(?4)*M_PI/8
               obj.alpha +:= BINARY_RAND*(?4)*M_PI/8
               }
            "7"|"8": {
               obj.val +:= BINARY_RAND*?10
               }
            "9"|"10": {
               obj.x +:= BINARY_RAND*?10
               obj.y +:= BINARY_RAND*?10
               }
            }
         }
```

```
      #
      # Calculate approximate fps, &time is in ms
      # To be more accurate, use a higher precision timer
      #
      time1 := &time
      Refresh()
      time2 := &time
      time := time2 - time1
      if time = 0 then time := 1
      fps := 1000.0/(time2-time1)

      #
      # Keep a rolling average
      #
      rolling_avg := 0
      if i > max_num then i := 0
      if *fps_list < max_num then put(fps_list, fps)
      else fps_list[i] := fps
      every rolling_avg +:= !fps_list
      rolling_avg /:= *fps_list
      fps_obj.s := fps_obj_shadow.s := string(integer(rolling_avg))
      i+:=1
      }
    }
end

procedure rects(n,objs,fill)
  local DrawProc
  width := WAttrib("width")
  height := WAttrib("height")

  if \fill then DrawProc := FillRectangle
  else DrawProc := DrawRectangle

  every i := 1 to n do {
    x := ?width - 1
    y := ?height - 1
    wid := ?(width/2) + 10
    ht := ?(height/2) + 10

    if mode == "pattern" then
      WAttrib("pattern="||?patterns, "fg="||?alphas||" "||?colors)
    else
      WAttrib("fg="||?alphas||" "||?colors)
    DrawProc(x, y, wid, ht)
    put(objs, WindowContents()[-1])
    }
end

procedure arcs(n,objs,fill)
  local DrawProc
  width := WAttrib("width")
  height := WAttrib("height")

  if \fill then DrawProc := FillArc
  else DrawProc := DrawArc

  every i := 1 to n do {
    x := ?width - 1
    y := ?height - 1
    wid := ?(width/2) + 10
    ht := ?(height/2) + 10

    if mode == "pattern" then
      WAttrib("pattern="||?patterns, "fg="||?alphas||" "||?colors)
```

```
      else
         WAttrib("fg="||?alphas||" "||?colors)
      DrawProc(x, y, wid, ht)
      put(objs, WindowContents()[-1])
      }
end

procedure circles(n,objs,fill)
   local DrawProc
   width := WAttrib("width")
   height := WAttrib("height")

   if \fill then DrawProc := FillCircle
   else DrawProc := DrawCircle

   every i := 1 to n do {
      x := ?width - 1
      y := ?height - 1
      r := ?((width+height)/8) + 5

      if mode == "pattern" then
         WAttrib("pattern="||?patterns, "fg="||?alphas||" "||?colors)
      else
         WAttrib("fg="||?alphas||" "||?colors)
      DrawProc(x, y, r)
      put(objs, WindowContents()[-1])
      }
end

procedure polygons(n,objs,fill)
   width := WAttrib("width")
   height := WAttrib("height")
   range := 100
   minpoints := 10

   if \fill then DrawProc := FillPolygon
   else DrawProc := DrawPolygon

   every i := 1 to n do {
      x := ?width - 1
      y := ?height - 1
      r := ?range + 5

      if mode == "pattern" then
         WAttrib("pattern="||?patterns, "fg="||?alphas||" "||?colors)
      else
         WAttrib("fg="||?alphas||" "||?colors)

      WAttrib("dx=0","dy=0")
      put(objs, WindowContents()[-2])
      put(objs, WindowContents()[-1])
      DrawProc(x, y, x+2*r, y, x+0.375*r, y+1.25*r, x+r, y-0.75*r, x+1.625*r, y+1.25*r)
      }
end

procedure drawstrings(n,objs)
   width := WAttrib("width")
   height := WAttrib("height")
   fht_range := 50
   fontlist := ["AvantGarde", "Bookman", "Charter", "Courier", "Gill Sans",
                "Helvetica", "Lucida Bright", "Lucida Sans",
                "New Century Schoolbook", "Palatino", "Rockwell", "Times"]
   attlist := ["", "bold", "italic", "bold,italic"]
   every i := 1 to n do {
      fheight := ?fht_range + 8
```

```
        x := ?(width) - 1
        y := ?(height-fheight) - 1

        WAttrib("fg="||?alphas||"  "||?colors)
        Font(?fontlist||","||fheight||","||?attlist)
        DrawString(x,y,text)
        put(objs, WindowContents()[-1])
        }

end

procedure copyareas(n,objs)
    width := WAttrib("width")
    height := WAttrib("height")
    im_wid := im_ht := 256
    x := y := 0

    ReadImage("unicon.jpg",x,y) | write("failed to write image")
    if n = 1 then return
    every i := 2 to n do {
        x_next := ?(width-im_wid) - 1
        y_next := ?(height-im_ht) - 1
        CopyArea(x, y, im_wid, im_ht, x_next, y_next)
        put(objs, WindowContents()[-1])
        x := x_next
        y := y_next
        }
end
```

Listing C.2: Source code for **animation.icn**