A Functional Approach to Graphics Programming and Modeling

by

S. Ramakrishnan

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Science

APPROVED:

[Signatures]
Adrienne Bloss (Chairperson)
James Arthur (co-Chair)
Osman Balci
Clifford Shaffer

September, 1992
Blacksburg, Virginia 24061
A Functional Approach to Graphics Programming and Modeling

by
S. Ramakrishnan

Committee Chairperson
Dr. Adrienne G. Bloss
Department of Computer Science

(ABSTRACT)

The FunGraf project was commenced with two broad aims: to explore the suitability of functional programming for graphics and to test the usability of the functional I/O models that have evolved recently. Because practical graphics applications tend to be complex and difficult to maintain, it is important to explore any avenue by which this complexity may be reduced. The declarative nature of image synthesis makes it reasonable to apply functional programming principles to this area, opening up avenues for formal program proving techniques.

We define a structured type hierarchy suitable for graphics programming in Haskell. The four geometric objects, point, line, arc and text are available as primitives using which arbitrarily complex pictures can be defined. Using Haskell, the programmer may define new types to suit the requirements of the application. The primitives are in turn categorized as two and three dimensional vectors. These two categories are respectively defined as components of the two and three dimensional picture segments. Picture segmentation is a useful technique in writing graphics applications. We have defined picture segmentation in 2D and 3D spaces. However, our implementation includes only the structures pertaining to 2D graphics. We also define basic workstation management operations. Using the continuation model, we extend the Haskell I/O framework to include functional interfaces to support picture display and the five standard graphics logical devices: locator, pick, stroke, valuator and string input. We use this framework to develop programs to illustrate how common graphics tasks may be implemented in a purely functional style.

This project highlights the advantages and disadvantages of using functional programming for graphics. While advantages of enhanced readability are difficult to document, programs are well structured and the absence of global variables greatly limits their complexity. The continuation I/O model was found adequate for graphics computation. Disadvantages of using functional programming for graphics include the problems caused by lazy evaluation in user interfaces and the possible degradation in performance of the system due to garbage collection.
Acknowledgements

The completion of this project would not have been possible without the useful ideas and suggestions put forth by Dr. Adrienne Bloss. I thank Dr. Cliff Shaffer for his useful inputs during development of the basic ideas. I am indebted to Dr. Osman Balcı for his constant encouragement to me during the past two years.

Many thanks to Woody Farley of Industrial and Systems Engineering Department for taking time to correct the draft.
# TABLE OF CONTENTS

1 Introduction 1

1.1 Overview 1

1.2 Goals of the Project 2

1.3 Thesis Overview 3

2 Other Unconventional Approaches to Graphics 5

2.1 Introduction 5

2.2 Object-Oriented Graphics Systems 5

2.3 Functional Graphics Systems 6

2.4 Contrasting the Functional Programming Project with Similar Efforts in the Past 8

3 Structure of FunGraf System 9

3.1 Introduction 9

3.2 An Introduction to Haskell 9

3.3 FunGraf Data Types 14

3.3.1 Primitive Graphics Types 14

3.3.1.1 Point Primitive 17

3.3.1.2 Line Primitive 18

3.3.1.3 Arc Primitive 23

3.3.1.4 Label Primitive 27

(iv)
3.3.2 Generalized Vector Primitive 29
3.3.3 Composite Types 30
3.4 Picture Modules 33
3.4.1 Picture Module in the Two Dimensional Space: Frame 33
3.5 Extension of Haskell Input-Output System for Graphics 37
3.5.1 Commonly Used Keywords in Graphics 37
3.5.2 A Framework for Graphics Input-Output 42
3.6 Illustration of Common User Interface Tasks Using the FunGraf Primitives 49

4 Notes on Implementation 56
4.1 Purpose of this Chapter 56
4.2 General View of the System Structure 56
4.3 Implementation of Graphics Input-Output Framework 62
4.3.1 Workstation Related Operations 65
4.3.2 Locator Related Operations 68
4.3.3 Implementation of Stroke Device 70
4.3.4 Valuator Related Operations 71
4.3.5 Keyboard Related Operations 72
4.3.6 Implementation of Frame Display Operation 73
4.4 A Note on the Implementation of Label Type 76

5 Conclusions and Future Work 77
5.1 Summary of Results 77
5.2 Concluding Remarks  77
  
  5.2.1 Readability and Structure of Programs  78
  
  5.2.2 Problems Caused by Lazy Evaluation in Graphical User Interfaces  78
  
  5.2.3 Effect of Garbage Collection and Software Layering on the Performance  82
  
5.3 Future Work  84
  
  5.3.1 Graphics Input-Output Using Stream Model and Systems Model  84
  
  5.3.2 Use of Boolean Operations and Application to 3D Graphics  85

References  86

Appendix A: Formal Description  87

Appendix B: Illustrative Programs  99
List of Illustrations

3.1 Auxiliary definitions for I/O framework 15
3.2 Continuation I/O model 16
3.3 Operations on the Point2 object 19
3.4 Operations on the Point3 object 20
3.5 Operations on the two dimensional line object 22
3.6 Operations on the three dimensional line object 24
3.7 Operations on the two dimensional arc object 26
3.8 Operations on the Label object 28
3.9 Operations on the generalized two dimensional vector 31
3.10 Examples of Frame objects 35
3.11 Operations on the Frame object 38
3.12 Channel Attributes 44
3.13 Repetition of actions using continuation I/O model and recursion 53
4.1 Layer Structure 57
4.2 List structures in Haskell and T 60
4.3 Interconversion of Haskell and T list structures 61
4.4 Establishing T-C interface 63
4.5 Implementation of graphics I/O calls 64
4.6 Implementation of the Workstation Object 66
5.1 Need for an explicit evaluation to sidestep lazy evaluation 80
5.2 Freezing of Display during Garbage Collection 83
B-1 Working of SCALE.HAS 107

B-2 Execution of RUBBERBAND.HAS 110

B-3 Using the String Input Device (STRDEVICE.HAS) 113

B-4 Painting Using the Locator Device (PAINTBRUSH.HAS) 116

B-5 (a) Main Menu 119
(b) Menu of Picture Segments 120
(c) User selects a house from the House Menu 121
(d) User is asked to select a pole from the Pole Menu 122
(e) Addition of a rotated car from the Vehicle Menu 123
(f) Creation of a composite segment from the Composite Menu 124
(g) The canvas after the addition of the composite segment 125
(h) The Final Picture 126
Chapter I

Introduction

1.1 Overview

John Backus in his Turing award lecture [Backus 1977] strongly advocates the use of the functional programming paradigm. Imperative programming languages closely reflect the underlying von Neumann architecture; they assume an implicit data structure called the 'memory' and define a program as a sequence of operations on this data structure. Major problems associated with this model include the effect of implicit operations (side-effects) and the sequential nature of the operations. Side effects make the maintenance of large programs difficult. Sequential operations make the construction of formal program proving methods difficult. In contrast to this, functional languages provide a set of primitive functions and a method of combining these functions to form complex programs. The main advantage of this paradigm is the lack of side effects, which considerably improves the readability and hence verifiability of the programs. Further, lazy evaluation, which is a notable feature of functional programming, can be used to write efficient and elegant programs.

Computer graphics is assuming greater significance in view of its increased application to newer areas. Graphics software tends to be complex and unwieldy despite the use of graphics functional standards (GFS), in which a library of functions provides to the graphics programmer a standard set of drawing, input-output and workstation management operations in a hardware-independent way. However, research in the area of programming language design has suggested tools to make the design of complex programs easier. These tools include modularization, data abstraction and functional abstraction, all of which are emphasized in the declarative style of programming. Declarative programming is naturally applicable to image synthesis [Foley and van Dam 1990] because any picture is merely a declaration: a line is declared by its end points, a rectangle by its vertices, a circle by its center and radius and so on. The advent of raster graphics has made image synthesis procedural: the algorithms of raster graphics are concerned with how to draw rather than what to draw. The burden of having to evaluate the position of each point on the image has been thrust on the programmer, while the ideal is to have the programmer deal only with the declaration of the image and have the system evaluate the rest of the image. The complexity in graphics software can be reduced by making
mechanisms less procedural and more declarative. Functional programming is one way of doing this.

Past research in the area of image synthesis in graphics has focused heavily on the development of fast algorithms that provide a visualization of given data. The development of the SIMD/MIMD class of graphics workstations has triggered research into parallel algorithms for operations such as rendering, hidden line/surface removal, ray tracing, etc. At a higher level, structured image synthesis such as that required for computer-aided design was supported in theory (e.g. CSG Trees, Euler representation, [Mantyla 1988]). For such purposes, the graphics functional standards provide segmented image description (e.g. GKS) and even hierarchical segmented image description (e.g. PHIGS). However, little or no emphasis has been placed on programming these methodologies, with the tacit assumption that some procedural language such as FORTRAN, Pascal or C would be used. These algorithms when implemented in such languages maintain their execution efficiency but lose their maintainability owing to the lack of structure in the language [Arya 1986]. This is evident in large CAD programs meant for the design of complex systems. Such programs written in C or Pascal are notoriously difficult to maintain. It is our belief that the use of functional programming techniques would considerably reduce the complexity of large software systems.

Another problem in conventional GFSs is the definition of redundant operations. Typically, a GFS provides drawing operations in two domains: for two dimensional entities and for three dimensional entities. From a higher level, the functions perform the same task, differing only in the details of how the task is performed on 2D and 3D domains. If the drawing operation can be overloaded on two and three dimensional entities, this duplication can be avoided. While this project does not implement this overloading, it shows how it could be implemented given the appropriate language constructs.

1.2 Goals of the Project

This project is aimed at developing a functional approach to computer graphics programming and modeling, based on the functional paradigm proposed by Backus. Since computer graphics is applied in critical areas, reducing the complexity of graphics software is a worthwhile goal. Functional programming is one of the available avenues. In this project, we focus on the pros and cons of applying functional programming to graphics and solutions to some of the problems. We introduce functional programming at the lowest level of programming for primitive functions. The style is carried to higher levels where we deal with picture modules, where the real advantage of functional programming begins to become obvious.
Models of real life entities are implemented functionally using abstract data types (ADT). The readability and verifiability of programs written in this framework are demonstrated through a simple landscape design program.

From the viewpoint of the functional programming community, an important goal of this project is to study the efficiency and programmer acceptability of the continuation I/O model [Hudak and Sundaresh 1989]. Side effect is the term used in functional programming to describe the change in the environment of a program caused by the execution of a function or a procedure. While the ideal of functional programming is non-side-effecting functions, the very nature of input-output is based on side-effects. The application of functional programming to interactive computer graphics has been hindered by the problem of performing I/O in a purely functional style. However, several models of purely functional I/O have recently evolved. These include:

1. the stream model
2. the continuation model and
3. the system model.

These models are described in greater detail later in this report. We found the continuation I/O mode to be adequate for the requirements of interactive graphics. Since graphics is I/O intensive, it forms an appropriate stage to test the model.

1.3 Thesis Overview

The following is a brief overview of this thesis. This chapter forms the introduction and prelude to the thesis.

Chapter 2 describes the conventional approaches to computer graphics and the efforts in the past to use object oriented programming and functional programming in graphics. This chapter also outlines in what respects the FunGraf project is different from similar efforts in the past.

Chapter 3 describes the basic structures and features of FunGraf. We start with a discussion of the primitive types and build more complex types required for graphics. Aspects of functional input-output as relevant to our system are explained. Finally, sample programs are given to illustrate the use of this system.

Chapter 4 presents the implementation aspects of the project. Description of the implementation of the type structures and graphics I/O functions is given here.

Chapter 5 forms the conclusion, outlining promising areas that were left unexplored in this
project. These are suggested as a future course of research in this area. Our impressions on the expressiveness of the continuation input-output model are described.

In appendices A and B, we present a brief and formal description of FunGraf. Also presented are programs and their graphical outputs, illustrating how programs may be written in this framework to perform some commonly used graphics operations.
Chapter II

Other Unconventional Approaches to Graphics

2.1 Introduction

The following sections contrast our approach with other unconventional approaches to graphics taken in the past. Conventionally, graphics software has been designed and implemented in the framework of an imperative programming language. Graphics functional standards (GFS) did not offer features such as classes, inheritance and referential transparency that aid the design of robust and readable programs (a function is said to be referentially transparent if it returns the same result for the same arguments under any circumstance). Further, the design of the functional standard was such that it forced the users of the standard to violate referential transparency (e.g., C binding of GKS). The software built using such GFS implementations is often complex and difficult to maintain.

There are mainly two unconventional paradigms with which graphics systems were built in the past: object-oriented graphics systems and functional graphics systems.

The first departure from conventional systems was initiated when object-oriented approaches were used in the design of user interfaces and modeling software. PHIGS [Clark 1992], albeit not an object-oriented system, was the first GFS to provide the feature crucial to the modeling of real-life objects: hierarchy. Later, ideas from the functional programming paradigm were introduced to graphics when Henderson [1982] suggested a functional approach to the description of geometric objects. While Henderson merely outlined how primitive graphics entities could be described functionally, Arya's work [1989] formed a major application of functional programming to animation.

The following sections describe the major efforts at designing graphics systems that formed a significant departure from the conventional systems. The concluding sections of this chapter contrast these systems with FunGraf and describe what this project contributes to the research in this area.

2.2 Object Oriented Graphics Systems

User interface design was the first area in computer graphics in which object oriented style
was used. Object oriented user interfaces provide screen controls as objects defined in a class hierarchy that may be used as provided or that can be customized to be specific to the application. Such systems were available on a small scale and their use was not widespread. A notable exception was the Mac Toolbox toolkit available on Apple Macintosh computers. In recent years, however, there has been a major effort at using object-oriented programming for the design of commercial grade user interfaces with the development of the X Intrinsics by the MIT X Consortium [Nye and O'Reilly 1990]. Commonly referred to as Xt, X Intrinsics is a standard established by the X Consortium that provides an object-oriented programming style in C. The purpose of Xt is to simplify the development of user interfaces by laying down mechanisms for the creation, extension and use of user interface elements such as pushbuttons, scrollbars, menus, dialog boxes so that the programmer need only integrate them with the application code to arrive at the desired user interface. In this mechanism, Xt defines a top level object class called Core. Core is a class that defines the most basic properties that any user interface element requires. More specific classes are defined as the subclasses of Core. Such classes are termed widget classes and an instance of a widget class is called a widget. Xt also defines the mechanisms for defining subclasses of existing objects. Designers of Xt did not develop a fixed set of components with a predefined look and feel. Instead, they created a general mechanism for producing reusable user interface components so that these components can be built either by the application programmer himself or by lower-level "object programmers" and then collected in libraries for use by the application. Motif is one such library [Heller 1991].

A major contribution of Xt to user interface development technology is the potential for software reuse. Once a widget with certain screen properties has been defined and implemented, it can be reused in any other software or even extended to a new class.

The developers at NeXT Computer Inc have launched a noteworthy effort at using object-oriented programming to build commercial grade user interfaces. The NeXT workstation provides a development environment for user interfaces based on an object-oriented style similar to X [NeXT 1991]. The NeXTStep system provides a set of readymade screen objects such as pushbuttons, menus and scrollbars that have predefined behavior. Such objects can be either directly included in the application or tailored to suit the specific needs of the application. Each object interacts with the application through an "action" routine that the programmer supplies while including the object in the application.

2.3 Functional Graphics Systems

The use of functional programming in computer graphics has not been as widespread as the
use of object-oriented programming. One reason for this is the fact that until recently a pure functional programming language with a framework to support functional I/O was not available. However, in the literature reported thus far, there have been notably two instances in which functional programming was used in graphics or a related area. The first instance was the description of functional geometry by Henderson [1982]. The other major application was the description of a functional animation system by Arya [1988]. With the development of Haskell in the past two years, functional programming is bound to become more acceptable to the graphics community.

The earliest known work related to the application of functional programming to graphics is the research by Henderson [1982]. Henderson suggests a simple method to describe geometric forms in a functional style. He proposes a type called picture to represent an arbitrary picture. A picture is composed of an arbitrary number of line segments defined on an integer coordinate system. Consequently an arbitrarily irregular figure can be represented using this data type. Henderson further defines operations such as rotation, scaling, juxtaposition (positioning a picture relative to another) and overlaying on pictures. The operations on pictures are defined functionally in terms of the corresponding operations on line segments. Finally, Henderson uses examples to illustrate how complex patterns could be described using the line segment alone as the primitive.

A more recent research involves the application of functional programming to computer animation [Arya 1989]. Arya's work is a serious effort using functional programming to produce animation sequences for commercial use. An increasing problem in the computer industry is that of preparing prototypes of animation sequences. Conventional programming environments use imperative programming languages such as C and result in the development of programs that are notoriously difficult to change. Arya's paper demonstrates how the functional approach leads to a fresh perspective on the problem. Functional programming allows for the development of a system that is easy to use and that allows rapid prototyping of animation sequences. The system provides a compact set of primitive operations over picture sequences and illustrates the use of these operations as a basis for an animation system. Higher order functions are used to construct a variety of tools that lead to a highly concise functional script. Having laid the framework for the functional animation system, Arya goes on further to investigate the properties of the system. Formal arguments to support intuitions regarding behaviour of animation models are presented and a purely functional mechanism for synchronising animation sequences is put forward.
2.4 Contrasting the FunGraf Project with Similar Efforts in the Past

The FunGraf project was begun with the purpose of designing and implementing a practical graphics system. The aim was to develop a precursor to a purely functional graphics system with many of the capabilities of GKS and PHIGS. In building such a system, we had to work in a relatively unexplored area of functional programming: input-output. One achievement of the FunGraf project has been experimentation with recent proposals for functional I/O [Hudak and Sundaresh 1989] and the moulding of the continuation I/O technique to suit the needs of a practical graphics system. These issues have never been addressed in the past.

The FunGraf project has significantly extended previous work in this area. Henderson's research was a landmark in that he, for the first time, suggested the use of functional programming for defining pictures. However, Henderson did not address the issues that arise when trying to make such a system practical. An elegant and practically usable graphics system needs to define a rich set of extensible types to accommodate the variety of objects encountered by a graphics system and application program. Much more than the simple types suggested by Henderson. Geometric forms seem to naturally fall into a hierarchy of categories in which properties of objects in one level depend on those of the objects at the lower level. An object oriented type system providing classes and inheritance seems naturally suitable to such an application (please note that while Haskell does not provide classes, it does offer an overloading mechanism). Henderson also does not address the issues associated with building interactive graphics systems functionally. This brings up the question of functional I/O, which is addressed in the FunGraf project.

Arya's research is similar to the FunGraf project only in its definition of "graphics objects" similar to picture suggested by Henderson and frame, used extensively in this project. Arya's application area has been the narrow domain of computer animation and does not address the issues related to making functional programming suitable to building a practical graphics programming system.
Chapter III

Structure of FunGraf System

3.1 Introduction

The FunGraf framework is a software system that provides to the graphics programmer a subset of the functionality of a graphics functional standard. This subset includes a set of drawing operations, standard graphics I/O operations and basic workstation management operations. The system has been implemented using a purely functional language and hence provides all the advantages of functional programming. Features of functional programming such as referential transparency and absence of global data structures make it less difficult for building program verification tools. A pure functional language provides a set of primitives and a method of combining these functions to form complex programs. Additionally, the user may also define new classes specific to the application based on existing (primitive) classes. FunGraf combines all these features to provide to the user an extensible graphics programming system.

This chapter introduces the type definitions and I/O mechanisms that constitute the FunGraf framework. The basic types are borrowed from Haskell; the primitive graphics types are then defined using the basic types and ADT mechanism. The graphics I/O operations use the higher level types to implement the corresponding I/O operations. In the concluding section, we present example programs to give a flavor of programs written in this system.

The framework we propose for graphics and modeling is based on many features of the functional language paradigm that have been realized in Haskell. Not only has the system been implemented using Haskell, but a syntax similar to Haskell is used here to describe the system. Further, to understand the example program fragments, the user must be familiar with Haskell. Hence, in the next section, we describe the most important features of Haskell.

3.2 An Introduction to Haskell

Haskell is a general purpose, purely functional programming language whose features include higher-order functions, non-strict semantics, static polymorphic typing user-defined algebraic data types, pattern-matching and list comprehensions [Hudak and Wadler 1990]. Following is a brief description of main features of Haskell aimed at introducing the basic
features of the language to the reader.

Types

Haskell includes a rich set of primitive types. They are integer, real, character, lazy lists, boolean and tuples. Haskell provides mechanism for the user to be able to define his own types in terms of the existing types. These are termed user defined Types. User defined types are identified by tagging a type expression. For instance,

\[
data\ Stack\ a = \text{Empty} \mid \text{MkStack}\ a\ (\text{Stack}\ a)
\]

which defines a type \text{Stack}\ a, in terms of the type variable \(a\). Hence \text{Stack}\ a\ is a structure which is either empty (\text{Empty}) or is a collection of objects of type \(a\), tagged by the tag \text{MkStack}. The tag \text{MkStack} serves to identify the type of the object that follows (\text{Stack}).

Every Haskell function is defined with its type signature. The type signature of a function is the string that declares the types of the arguments to the function and the type of the data returned by the function. Hence a function \(f\) whose type signature is

\[f :: \text{Int} \to \text{Float} \to \text{Bool}\]

takes as argument an integer and a float. The function returns a boolean value.

An important feature of Haskell type system is the overloading mechanism called 'class'. Type classes are provided to extend the traditional Hindley-Milner polymorphic type system. This allows for a structured definition of overloaded functions and is the major technical innovation in Haskell. A class introduces a new type class and the overloaded operations that must be supported by any type that is an instance of that class. An instance declaration declares a type to be an instance of a class. An operation may be overloaded on a specified set of types by this definition. For example, the Haskell declarations to overload the operation \text{consume} two types \text{water} and \text{bread}, are as follows.

\[
\text{class Edible}\ a\ \text{where}
\text{consume} ::\ a\ \to\ \text{Boolean}
\]

This defines a 'general' type \text{Edible} which has an operation \text{consume} defined on it. Identifier \(a\) is a type variable used as a parameter in this definition. The following two declarations declare
the types on which consume is overloaded.

    instance Edible bread where
      consume = eat
    instance Edible milk where
      consume = drink

Note that eat is the version of consume defined on type bread and drink is the version of
consume for type milk. Note that 'classes' in Haskell are merely overloading mechanisms and
are different from classes in object-oriented languages such as C and Smalltalk.

Haskell class definitions allow only a single parameter. This has proved to be a limitation in
the definition of FunGraf (section 3.3.1.2).

**Special Features**

The following special features of Haskell deserve to be commented upon.

*Pattern matching* is used extensively in function definitions and expressions, intuitively
similar to Prolog predicates. For instance, the fragment

    case e of
      [0,_,_] -> True
      [_,_,_] -> False

This fragment evaluates to True if the first element of the list is an atom and its value is 0. The
symbol _ is used as a wildcard to match any item. Using this fragment, the expression [0, x, y ]
will evaluate to True, since the pattern [0, _, _] is defined before the pattern [_, _, _].

*Lazy evaluation* is a style of evaluation wherein an expression is not evaluated until its
value is immediately required. Unnecessary computations may be performed in conventional
programming due to strict evaluation as illustrated by the following example:

    i :: Int -> Int -> Int -> Int
    i x y z = if (x > 0) then y else y + z
If strict evaluation were used, the invocation \( i(1, 2, 5/3) \) would require the computation of \( 5/3 \) despite the fact that with the given values of \( x \) and \( y \), the value of the actual parameter corresponding to \( z \) is not required to evaluate the function. Further, programs will sometimes terminate under lazy evaluation that would fail to terminate (or give an error) under strict evaluation. Thus, in theory, efficiency of programs can be increased significantly by avoiding such redundant computations using lazy evaluation. In practice, however, lazy evaluation may not be efficient; its advantage lies in that it allows for a different (and often more declarative) programming style.

**Functional Input-Output**

While I/O can be described as another special feature of Haskell, the sheer significance of it calls for a special emphasis. Haskell provides three models [Hudak and Sundaresh 1989] for performing functional input-output: the Stream model, the Continuation model, and the System model. At the time of writing this report, the System model had not yet been implemented in the Yale Haskell version. Since we have chosen the Continuation model for our project, we describe it here in greater detail.

Under the continuation model, a Haskell program has the type

\[
[\text{Response}] \rightarrow [\text{Request}]
\]

where the program sends Requests to the operating system which in turn returns Responses to the program. A request can pertain to files (read file, write file, append file, delete file, query status), to I/O channels (read channel, write channel, append channel, query status) or to the graphics operations (open graphics workstation, display picture module, read locator, readvaluator, read pick device, read keyboard, close workstation). The operating system always returns to the program a response in response to a request sent in by the program. The response corresponding to a write request is always either Success or Failure. The additional information returned along with the Failure response identifies the reasons for the Failure. If a read request succeeds, the operating system returns the type of the data read in which can be one of Str (character string), Bin (numbers), Loc (mouse position) or Fr (Frame). The last two types are FunGraf types. If a read request fails for some reason, the operating system returns Failure as the response. As usual, the additional information accompanying this response identifies the reasons for the failure.

Every I/O function based on the continuation model has three main parts: (1) the function
performing the I/O operation, (2) the function that handles the failure of the operation (the failure continuation) and (3) the function that handles the success if the operation is successful (the success continuation) [Hudak and Sundaresh 1989]. The function that performs the I/O operation, takes as arguments the following entities:

1. the channel identification
2. optional arguments (attributes values, in case of openWorkStation call and data to be output, in case of output functions)
3. the failure continuation function
4. the success continuation function.

The type of the function application is Dialogue (Figure 3.1). Note that Dialogue is the type of any function application that sends requests (Request) to the operating system and receives responses (Response) in return. Any function f that has argument of type X and that performs input-output operations must have the type signature

\[ f :: X \rightarrow \text{Dialogue} \]

A function f that has the success continuation S and failure continuation F and that operates on the channel channel is invoked as

\[ f \text{ channel} \ [ \text{optional arguments} ] \ F \ S \]

If f were an input function that expected to read a data element of type T from channel, then the success continuation S would be of the type

\[ S :: T \rightarrow \text{Dialogue} \]

If, however, f were outputting data on channel channel, then the success continuation would not need any arguments and in this event its type would be

\[ S :: \text{Dialogue} \]

The failure continuation, however, has the same type in either case. The failure continuation
takes in an argument describing the type of the error and returns the type `Dialogue`. Hence the type of failure continuation is

\[ F :: \text{IOError} \rightarrow \text{Dialogue} \]

The type `IOError` is a Haskell defined type that describes the type of the error encountered in performing the I/O operation (such as write error, read error, format error and so on). The working of the continuation I/O model has been graphically illustrated in Figure 3.2. For a more detailed explanation of the continuation style, the reader is referred to [Hudak and Sundaresh 1989].

### 3.3 FunGraf Data Types

The FunGraf type framework comprises three types: (1) the basic types required for general purpose computation, (2) the primitive type classes representing the basic geometric objects and (3) the composite types that are new types that are complexes of existing types. The basic types are borrowed from Haskell and were described earlier. In the following sections, we describe the primitive type structures found necessary to represent geometric forms and the mechanism for defining composite types. These structures are not built into Haskell but have been defined using the mechanism provided by Haskell. The user may build specific types based on these primitive types using Haskell mechanisms. This is illustrated in later sections.

#### 3.3.1 Primitive Graphics Types

FunGraf defines pictures in terms of the four basic geometric forms: points, lines, arcs and text strings. Consequently, the types corresponding to these geometric forms are provided as primitives. These are, namely, `Point2`, `Line2`, `Arc2` and `Label` in the two dimensional domain. The corresponding primitives in the three dimensional domain are `Point3`, `Line3` and `Arc3`. Text strings (type `Label`) are not defined in the three dimensional space. The rest of the types that are needed for graphics computation are defined in terms of these structures hierarchically. In the following sections, type classes corresponding to the four primitives are defined to indicate the similarities between 2D and 3D types. However, these classes are not used in the implementation because of the problem discussed in section 3.3.1.2.

For each FunGraf data type, we define a *null entity* of that type. Null entities are required
type Channel = Int

type Dialogue = [Response] -> [Request]

type SuccCont = Dialogue

type FailCont = IOError -> Dialogue

type StrCont = String -> Dialogue

type WsCont = Channel -> Dialogue

type LocCont = Point -> Dialogue

type PickCont = Frame -> Dialogue

type ValuatorCont = Float -> Dialogue

type StrokeCont = [Point2] -> Dialogue

Figure 3.1: Auxiliary Definitions for I/O Framework
[a] Input

Input Successful

Input Call

Call
Success Continuation
with input as argument

Call
Failure Continuation
with error description

[b] Output

Output Successful

Output Call

Call
Success Continuation
with no arguments

Call
Failure Continuation
with error description

Figure 3.2 Continuation I/O Model
to establish the closure of operations on the domain and are generally used to represent invalid or 'empty' result of the type.

3.3.1.1 Point Primitive

Point is a type class representing the fundamental geometric object of the system, namely a point, in the two and three dimensional coordinate space. While a two dimensional point is defined as an ordered pair of two floating point numbers, a three dimensional point is an ordered triple:

```plaintext
data Point2 = EmptyPoint2 | Pt2 Float Float
  data Point3 = EmptyPoint3 | Pt3 Float Float Float
```

The tags Pt2 and Pt3 serve to identify the user defined types. They are also known as type constructors. The null entities of the types Point2 and Point3 are respectively, EmptyPoint2 and EmptyPoint3.

Operations on Point2 are defined as follows (Figure 3.3). The functions that implement these operations are defined using the pattern matching feature of a purely functional language. The _ character is a pattern that matches any argument (a don't care feature).

* A two dimensional point is scaled using a real number by scaling its x and y coordinates by the factor. Scaling the null entity by any factor returns the null entity.
* A two dimensional point is displaced by another two dimensional point by adding the corresponding x and y coordinates of the points. Again, moving the null entity results in a null entity.
* Rotating a two dimensional point (Pt2 x y) by an angle alfa about a pivot point (Pt2 px py) is done by displacing the point obtained by rotating the point (Pt2 (x - px) (y - py)) about origin by the angle alfa, by the point (Pt2 px py).

The Point class is defined as follows. Operations of scaling, rotation and translation are defined on all instances of this class. This definition uses a as a type variable.

```plaintext
class Point a where
  scalePoint :: a -> Float -> a
  movePoint :: a -> a -> a
  rotatePoint :: a -> a -> a -> a
```

17
Function `scalePoint` takes a point and a real number and returns a point scaled by that value. Function `movePoint` takes two points and returns a point which is the first argument displaced by the second argument. Function `rotatePoint` returns the point obtained by rotating the given point by the given angle about the given pivot point. The instance mechanism is used to define points in different dimensions. We define, as instances of the class `Point`, the two dimensional point (`Point2`) and three dimensional point (`Point3`).

```haskell
instance Point Point2 where
    scalePoint = scalePt2
    movePoint = movePt2
    rotatePoint = rotatePt2
```

Scaling, translation and rotation operations on type `Point3` are analogous to those on the type `Point2`. The operations defined on `Point3` affect all the three component `Float` elements of the `Point3` type.

The following declaration includes `Point3` in the class `Point`:

```haskell
instance Point Point3 where
    scalePoint = scalePt3
    movePoint = movePt3
    rotatePoint = rotatePt3
```

### 3.3.1.2 Line Primitive

`Line` is another fundamental vector type of the system and is probably the most frequently used vector when describing pictures. The type class `Line` represents lines in the two and three dimensional coordinate space. As in the case of class `Point`, the operations scaling, rotation and translation are defined on all instances of this class. `Line` is defined as follows:

```haskell
class Line a where
    scaleLine :: a -> Float -> a
    moveLine :: a -> Float -> a
    rotateLine :: a -> Float -> Point -> a
    endPoints :: a -> (Point,Point)
```
(a)

-- Scale a 2D point by a scalar factor.

\[
\text{scalePt2 :: Point2 -> Float -> Point2}
\]

\[
\text{scalePt2 EmptyPoint2 _ = EmptyPoint2}
\]

\[
\text{scalePt2 \ (Pt2 \ x \ y) \ s = Pt2 \ s \cdot x \ s \cdot y}
\]

(b)

\[
\text{movePt2 :: Point2 -> Point2 -> Point2}
\]

\[
\text{movePt2 EmptyPoint2 _ = EmptyPoint2}
\]

\[
\text{movePt2 \ (Pt2 \ x \ y) \ (Pt2 \ u \ v) = Pt2 \ (x + u) \ (y + v)}
\]

(c)

-- Rotate the given point by the given angle about the given pivot.

\[
\text{rotatePt2 :: Point2 -> Float -> Point2 -> Point2}
\]

\[
\text{rotatePt2 \ (Pt2 \ x \ y) \ alfa \ (Pt2 \ px \ py) =}
\]

Figure 3.3 Operations on the Point2 object
   (a) Scaling  (b) Translation and
   (c) Rotation
(a)  
-- Scale a three dimensional point by a scalar factor.

scalePt3 :: Point3 -> Float -> Point3

scalePt3 EmptyPoint3 _ = EmptyPoint3

scalePt3 (Pt3 x y z) s = Pt3 (s * x) (s * y) (s * z)

(b)  
movePt3 :: Point3 -> Point3 -> Point3

movePt3 EmptyPoint3 _ = EmptyPoint3

movePt3 (Pt3 x y z) (Pt3 u v w) = Pt3 (x + u) (y + v) (z + w)

(c)  
-- Rotate the given point by the given angle about the given pivot.

rotatePt3 :: Point3 -> Float -> Point3 -> Point3

rotatePt3 (Pt3 x y z) alfa (Pt3 px py pz) =

Figure 3.4  Operations on the Point3 object
(a) Scaling  (b) Translation and
(c) Rotation
The scaling, moving and rotation operation have the meaning as for the Point class. The operation **endPoints** returns the end points of the argument line as a pair. This definition brings out a limitation of the Haskell overloading mechanism (section 3.2). The type of data returned by the function **endPoint** depends on the particular instance of the class. However, there is no way to indicate that information in this definition. This problem could be solved if Haskell allowed more than one parameter in a class definition as follows:

```hs
class Line a b where
    endPoints :: a -> (b, b)
```

However, this is not a legal construct in the current version of the language.

Any practical graphics programming system requires the use of lines in two and three dimensional spaces. The instance mechanism is again used to define these types as the members of the class **Line**.

Instances of the class **Line** are two dimensional lines (**Line2**) and three dimensional lines (**Line3**). A two dimensional line is an ordered pair of two **Point2** objects and a three dimensional point is an ordered pair of two **Point3** objects:

```hs
data Line2 = EmptyLine2 | MkLine2 Point2 Point2
```

Thus, a two dimensional line is either a null entity (**EmptyLine2**) or an ordered pair of two **Point2** objects. The tag **MkLine2** serves to identify the type. Similarly we define the three dimensional line:

```hs
data Line3 = EmptyLine3 | MkLine3 Point3 Point3
```

The scaling of a two dimensional line is defined in terms of the scaling of its endpoints (Figure 3.5 [a]). The scaling of **Line2** is defined in terms of the corresponding operation on its component (**Point2**) types. A two dimensional line is displaced by a two dimensional point by displacing its endpoints (Figure 3.5 [b]). A two dimensional line is rotated by rotating its endpoints (Figure 3.5 [c]). It is trivial to implement a function to return the endpoints (Figure 3.5 [d]).

Type **Line2** is included in the class **Line** by the following declaration:
(a) 
\[ \text{scaleLine2} :: \text{Line2} \rightarrow \text{Float} \rightarrow \text{Line2} \]
\[ \text{scaleLine2} \; \text{EmptyLine2} \; _{} = \text{EmptyLine2} \]
\[ \text{scaleLine2} \; (\text{MkLine2} \; \text{point1} \; \text{point2}) \; s = \]
\[ \text{MkLine2} \; (\text{scalePt2} \; \text{point1} \; s) \; (\text{scalePt2} \; \text{point2} \; s) \]

(b) 
\[ \text{moveLine2} :: \text{Line2} \rightarrow \text{Point2} \rightarrow \text{Line2} \]
\[ \text{moveLine2} \; \text{EmptyLine2} \; _{} = \text{EmptyLine2} \]
\[ \text{moveLine2} \; (\text{MkLine2} \; \text{point1} \; \text{point2}) \; \text{point} = \]
\[ \text{MkLine2} \; (\text{movePt2} \; \text{point1} \; \text{point}) \; (\text{movePt2} \; \text{point2} \; \text{point}) \]

(c) 
\[ \text{rotateLine2} :: \text{Line2} \rightarrow \text{Float} \rightarrow \text{Point2} \rightarrow \text{Line2} \]
\[ \text{rotateLine2} \; \text{EmptyLine2} \; _{} \; _{} = \text{EmptyLine2} \]
\[ \text{rotateLine2} \; \text{line} \; 0.0 \; _{} = \text{line} \]
\[ \text{rotateLine2} \; (\text{MkLine2} \; \text{point1} \; \text{point2}) \; \text{angle} \; \text{point} = \]
\[ \text{MkLine2} \; (\text{rotatePt2} \; \text{point1} \; \text{angle} \; \text{point}) \]
\[ (\text{rotatePt2} \; \text{point2} \; \text{angle} \; \text{point}) \]

(d) 
\[ \text{endPoints2} :: \text{Line2} \rightarrow (\text{Point2},\text{Point2}) \]
\[ \text{endPoints2} \; \text{EmptyLine2} = (\text{EmptyPoint2},\text{EmptyPoint2}) \]
\[ \text{endPoints2} \; (\text{MkLine2} \; \text{point1} \; \text{point2}) = (\text{point1},\text{point2}) \]

Figure 3.5  Operations on the two dimensional line object
(a) Scaling  (b) Translation and (c) Rotation
and (d) Retrieve endpoints
instance Line Line2 where
   scaleLine = scaleLine2
   moveLine = moveLine2
   rotateLine = rotateLine2
   endPoints = endPoints2

Operations on the three dimensional line (scaling, rotation, moving, endpoints) can be
defined on the lines of those implemented for the two dimensional line. A three dimensional line
is displaced by a point Point3 by displacing its endpoints (Figure 3.6 [b]). Rotation of a three
dimensional line is defined in terms of the rotation of its endpoints (Figure 3.6 [c]).

The type Line3 is a member of the type class Line and represents lines in the three
dimensional space. The following fragment formally declares this membership:

instance Line Line3 where
   scaleLine = scaleLine3
   moveLine = moveLine3
   rotateLine = rotateLine3
   endPoints = endPoints3

Note that while the preliminary declarations have been made for the basic 3D objects, the
operations of projection and viewing have not been defined for these structures. The internals of
these definitions are not visible to the top-level programmer.

3.3.1.3 Arc Primitive

Arc is a primitive that facilitates description of curved shapes. This forms the third basic
primitive of the FunGraf system. To keep the system simple, we decided to include only arcs of
circles in the primitive set. An arc is defined by the following parameters:

1. its center (Point2 or Point3)
2. its radius (Float)
3. the positive angle subtended at the center by the 'first' end (this being defined as the
   end of the arc that is encountered last in a clockwise sweep of the arc) and the point at
   the 3 O' clock position on the circle containing the arc (Float).
4. the counterclockwise angle subtended by the two ends of the arc at the center (Float).
(a)

scaleLine3 :: Line3 -> Float -> Line3
scaleLine3 EmptyLine3 _ = EmptyLine3
scaleLine3 (MkLine3 point1 point2) s =
    MkLine3 (scalePt3 point1 s) (scalePt3 point2 s)

(b)

moveLine3 :: Line3 -> Point3 -> Line3
moveLine3 EmptyLine3 _ = EmptyLine3
moveLine3 (MkLine3 point1 point2) point =
    MkLine3 (movePt3 point1 point) (movePt3 point2 point)

(c)

rotateLine3 :: Line3 -> Float -> Point3 -> Line3
rotateLine3 EmptyLine3 _ _ = EmptyLine3
rotateLine3 line 0.0 _ = line
rotateLine3 (MkLine3 point1 point2) angle point =
    MkLine3 (rotatePt3 point1 angle point)
    (rotatePt3 point2 angle point)

(d)

depPoints3 :: Line3 -> (Point3,Point3)
depPoints3 EmptyLine3 = (EmptyPoint3,EmptyPoint3)
depPoints3 (MkLine3 point1 point2) = (point1,point2)

Figure 3.6 Operations on the three dimensional line object
(a) Scaling  (b) Translation and (c) Rotation
and (d) Retrieve endpoints
Similar to the primitives defined earlier, the operations on the arc class include scaling, rotation about a point, and translation. Additional operations could be defined on the primitive as necessary. The two and three dimensional version of the arc primitive are defined as follows.

```
data Arc2 = EmptyArc2 | MkArc2 Point2 Float Float Float
data Arc3 = EmptyArc3 | MkArc3 Point3 Float Float Float
```

EmptyArc2 and EmptyArc3 are the null entities of types Arc2 and Arc3 respectively. The tags MkArc2 and MkArc3 serve to identify the types. The operations on the arc types are defined as follows. An arc is scaled by scaling its center and the radius (Figure 3.7 [a]). Translation of an arc is defined in terms of the translation of its center alone (Figure 3.7 [b]). Rotation of an arc results in an arc whose center is obtained by rotating the center of the original arc and whose endpoints are obtained by rotating the endpoints of the given arc (Figure 3.7 [c]). These operations are similarly (scaleArc3, moveArc3, rotateArc3) defined on arcs in three dimensional space.

The Arc class definition is given below:

```
class Arc a where
  scaleArc :: a -> Float -> a
  moveArc :: a -> Float -> a
  rotateArc :: a -> Float -> Point -> a
```

The types Arc2 is made a member of the Arc class by the declarations:

```
instance Arc Arc2 where
  scaleArc = scaleArc2
  moveArc = moveArc2
  rotateArc = rotateArc2
```

A very similar construct is used to declare Arc3 as an instance of class Arc:

```
instance Arc Arc3 where
  ( ... )
```
(a)

scaleArc2 :: Arc2 -> Float -> Arc2
scaleArc2 EmptyArc2 _ = EmptyArc2
scaleArc2 _ 0.0 = EmptyArc2
scaleArc2 (MkArc2 center radius angle sweep) s =
    MkArc2 (scalePt2 center s) s*radius angle sweep

(b)

moveArc2 :: Arc2 -> Point2 -> Arc2
moveArc2 EmptyArc2 _ = EmptyArc2
moveArc2 (MkArc2 center radius angle sweep) point =
    MkArc2 (movePt2 center point) radius angle sweep

(c)

rotateArc2 :: Arc2 -> Float -> Point2 -> Arc2
rotateArc2 EmptyArc2 _ _ = EmptyArc2
rotateArc2 arc 0.0 _ = arc
rotateArc2 (MkArc2 center radius angle sweep) alfa pivot =
    MkArc2 ncenter radius nangle sweep
    where
        ncenter = rotatePt2 center alfa pivot
        nangle = rotateAngle nangle center alfa pivot

Figure 3.7  Operations on the two dimensional arc object
(a) Scaling (b) Translation and (c) Rotation
3.3.1.4 Label Primitive

The ability to draw text in different styles is a feature mandatory to commercial quality graphics systems. While text can be described by the application programmer in terms of points, lines and arcs, doing this imposes on the application an unnecessary complexity. FunGraf provides the Label primitive to facilitate drawing of graphics text. Our system relies on the font implementation of the underlying platform, namely the X Window System, to implement this primitive. Hence to define a Label primitive, the user needs to specify the following three parameters: the text string, the window and the viewport. The window and viewport information is required in the label encapsulation to check if a given point lies in the label string. This is a direct result of the inelegant implementation of text strings in the underlying graphics system (X). This is discussed in greater detail in Chapter IV.

While a label class has been defined for the sake of uniformity, the label type exists in the system only in the two dimensional space. Further, owing to the raster implementation of fonts in X, the only geometric operation defined on Label primitive in Functional Graphics is the translation operation. Rotation and scaling are not defined. Vector fonts are likely to be made available in future versions of X. Hence we define a Label class in FunGraf despite the fact that the class has currently only one member.

The label class is defined as follows:

```
class LabelClass where
    moveGeneralLabel :: Label -> Point -> Label
```

A label in the two dimensional space is defined as follows:

```
data Label = EmptyLabel | MkLabel String Point Window2 Window2
```

The first component defines the string of the label. The second member defines the location of the origin of the font-string in the world coordinate space. The last two members respectively identify the window and viewport with which the vector is displayed on the workstation.

Label translation is defined by the function `moveLabel` (Figure 3.8 [a]). Due to the reasons discussed in the previous paragraphs, scaling and rotation of Labels are defined to be identity functions.

The label instance definition is as follows:
(a)

\[ \text{moveLabel} :: \text{Label} \rightarrow \text{Point2} \rightarrow \text{Label} \]

\[ \text{moveLabel} \ \text{EmptyLabel} \_ = \text{EmptyLabel} \]

\[ \text{moveLabel} \ \text{(MkLabel} \ s \ p \ w \ v) \ \text{to} = \text{MkLabel} \ s \ \text{(movePt2} \ p \ \text{to)} \ w \ v \]

(b)

\[ \text{scaleLabel} :: \text{Label} \rightarrow \text{Float} \rightarrow \text{Label} \]

\[ \text{scaleLabel} \ \text{label} \_ = \text{label} \]

(c)

\[ \text{rotateLabel} :: \text{Label} \rightarrow \text{Float} \rightarrow \text{Point} \rightarrow \text{Label} \]

\[ \text{rotateLabel} \ \text{label} \_ \_ = \text{label} \]

Figure 3.8  Operations on the Label object
(a) Scaling  (b) Translation and
(c) Rotation
instance LabelClass Label where
moveGeneralLabel = moveLabel

3.3.2 Generalized Vector Primitive

We have now defined the basic primitives of the type system. Before we use these primitives to define picture segmentation, it is necessary to introduce a pseudo-type, the generalized vector primitive. The generalized vector primitive is a pseudotype which represents one of the four basic primitives.

Consider the two dimensional space to begin with. For our purposes, we would like to define a generic type Vector2 to refer to any one of the four two dimensional primitive types ('vectors'). Intuitively, Vector2 can be thought of a type that represents a generalized two dimensional vector and is defined as follows:

data Vector2 = EmptyVector2
    | Pt2ToVector2 Point2
    | Line2ToVector2 Line2
    | Arc2ToVector2 Arc2
    | LabelToVector2 Label

EmptyVector2 is the null entity of the type Vector2. Pt2ToVector2, Line2ToVector2, Arc2ToVector2 and LabelToVector2 are tags that serve to identify the current type (i.e., Vector2) and the type of the data entity following the tag. For instance, the vector

vector = Pt2ToVector2 (Pt2 10.0 30.0)

represents the two dimensional point (10.0, 30.0). The three dimensional generalized vector, Vector3, can be defined similarly:

data Vector3 = EmptyVector3
    | Pt3ToVector3 Point3
    | Line3ToVector3 Line3
    | Arc3ToVector3 Arc3
    | LabelToVector3 Label
Operations on generalized vectors are defined in terms of the corresponding operations on the vectors constituting the type. The operations of scaling, rotation and translation are defined on Vector2 and Vector3. The scaling operation on a vector2 calls the scaling function of a point, line, arc or label depending on whether the vector2 is a point, a line, an arc or a label (Figure 3.9). Other two operations on vector2 are similarly defined. On the same lines, we have operations of scaling, rotation and translation defined on vector3 type objects also.

3.3.3 Composite Types

Composite graphics types define commonly used non-primitive graphics objects using the primitive types defined above. Examples of such types are circles, rectangles (windows), etc. The following examples illustrate some such types.

A circle can be defined using the primitive types as follows:

\[
\text{type Circle2} = \text{MkCircle2 Point2 Float}
\]

The first component identifies the center and the second, the radius. This synthesized datatype may be used as in the following function to find if a given point lies on a circle. \(\text{MkCircle2}\) is a tag that identifies the data type in expressions.

\[
\text{pointOnCircle} :: \text{Circle2} \rightarrow \text{Point2} \rightarrow \text{Bool}
\]

- if the given point lies on the given circle, return \(\text{True}\), else \(\text{False}\).

\[
\text{pointOnCircle} \ (\text{MkCircle2} \ (\text{Plt2} \ cx \ cy) \ r) \ (\text{Plt2} \ x \ y) = \]
\[
((x - cx)^2 + (y - cy)^2) = r^2
\]

Another example is the definition of a rectangular boundary, used as windows in graphics. A window may be defined as follows:

\[
\text{data Window2} = \text{MkWindow2 Point2 Point2}
\]

The first point identifies the lower left vertex and the second identifies the upper right vertex. The tag \(\text{MkWindow2}\) serves to identify the data type in expressions.
(a)

\[ \text{scaleVector2 :: Vector2 -> Float -> Vector2} \]

\[ \text{scaleVector2 EmptyVector2 _ = EmptyVector2} \]

\[ \text{scaleVector2 _ 0.0 = EmptyVector2} \]

\[ \text{scaleVector2 (Pt2ToVector2 point) factor =} \]

\[ \text{Pt2ToVector2 (scalePt2 point factor)} \]

\[ \text{scaleVector2 (Line2ToVector2 line) factor =} \]

\[ \text{Line2ToVector2 (scaleLine2 line factor)} \]

\[ \text{scaleVector2 (Arc2ToVector2 arc) factor =} \]

\[ \text{Arc2ToVector2 (scaleArc2 arc factor)} \]

\[ \text{scaleVector2 (LabelToVector2 label) factor =} \]

\[ \text{LabelToVector2 (scaleLabel label factor)} \]

(b)

-- Obtain the Vector2 obtained by translating the given Vector2 to
-- the given point.

\[ \text{moveVector2 :: Vector2 -> Point2 -> Vector2} \]

\[ \text{moveVector2 EmptyVector2 _ = EmptyVector2} \]

\[ \text{moveVector2 (Pt2ToVector2 point) to =} \]

\[ \text{Pt2ToVector2 (movePt2 point to)} \]

\[ \text{moveVector2 (Line2ToVector2 line) to =} \]

\[ \text{Line2ToVector2 (moveLine2 line to)} \]

\[ \text{moveVector2 (Arc2ToVector2 arc) to =} \]

\[ \text{Arc2ToVector2 (moveArc2 arc to)} \]

\[ \text{moveVector2 (LabelToVector2 label) to =} \]

\[ \text{LabelToVector2 (moveLabel label to)} \]

Figure 3.9 Operations on the generalized two dimensional vector (a) Scaling (b) Translation
-- Obtain the Vector2 by rotating anticlockwise the given
-- Vector2 by the given angle about the given pivot (Point2).
rotateVector2 :: Vector2 -> Float -> Point2 -> Vector2
rotateVector2 EmptyVector2 _ _ = EmptyVector2
rotateVector2 vector 0.0 _ = vector2
rotateVector2 (Pt2ToVector2 point) angle pivot =
    Pt2ToVector2 (rotatePt2 point angle pivot)
rotateVector2 (Line2ToVector2 line) angle pivot =
    Line2ToVector2 (rotateLine2 line angle pivot)
rotateVector2 (Arc2ToVector2 arc) angle pivot =
    Arc2ToVector2 (rotateArc2 arc angle pivot)
rotateVector2 (LabelToVector2 label) angle pivot =
    LabelToVector2 (rotateLabel label angle pivot)

Figure 3.9 (c) Rotation operation on the generalized two
dimensional vector
3.4 Picture Modules

Modular image synthesis has been in vogue for some time now. This feature of graphics functional standards is widely used in modeling and CAD applications. Many of the modeling techniques define a complex object in terms of union (glueing), subtraction (chipping off) and intersection of transformed primitive objects [Mantyla 1988]. Hence primitive objects are provided as picture modules which are used as building blocks in the construction of complex objects. A picture module is a collection of generalized vectors much in the same way that a procedure is a collection of executable statements in a computer program. When a set of primitives needs to be replicated a number of times in the graphics output, the programmer may define this collection as a picture and supply the name of the picture module to the GFS interface when it needs to be drawn. A variety of operations are defined on picture modules. All the operations defined on generalized vectors are defined on picture modules also. Further, the operations of replication and boolean operations such as union, intersection and subtraction can also be defined on this data type.

The FunGraf system defines picture modules in both two and three dimensional space. The module in two dimensional space is referred to as Frame. Its three dimensional counterpart is Scene. In the FunGraf system, picture modules are the least complex types that can be subject to graphics input (pick) and output (display) operations. Hence a vector must be encapsuled in a frame before it can be displayed.

3.4.1 Picture Module in Two Dimensional Space: Frame

A frame is a collection of two dimensional vectors that can be referred to by an identifier. The object in real life that a frame represents is a complete picture or a part of a picture. A frame can be used to build more complex frames using combining mechanisms. Mathematically, a frame is defined as follows:

1. the null entity, EmptyFrame, is a frame.
2. if v is a Vector2 and f is a frame, then
   \[
   \text{MkFrame} \; v \; f
   \]
   is a frame. The tag MkFrame serves to identify the frame type.
3. If f1 and f2 are frames, then so is their union, denoted by
   \[
   \text{UnionFrame} \; f1 \; f2
   \]
4. If f1 and f2 are frames, then so is their intersection, denoted by
   \[
   \text{intersectFrame} \; f1 \; f2
   \]
5. If \( f_1 \) and \( f_2 \) are frames, then so is the subtraction of \( f_2 \) from \( f_1 \), denoted by

\[
\text{FrameDifference } f_1 \ f_2
\]

While we have different representations of a given frame object, a particular representation assumes significance in I/O operations involving the frame. This representation, called the normal form, is defined as follows:

* **EmptyFrame** is in normal form
  * if the frame \( f \) is in normal form, then so is the frame

\[
\text{MkFrame } \text{vector } f
\]

where \( \text{vector} \) an object of type Vector2.

The normal form assumes significance in the definition of I/O operations (section 3.5.2). We now present a few examples of frames to illustrate the use of this data type in programming. The following examples demonstrate how the frame structure can be used for defining real life objects:

**Example 1**

The following frame definition corresponds to the image shown in Figure 3.10 [a]:

\[
\text{frame} = \text{MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 0.0) (Pt2 1.0 1.0)))}
\]

\[
(\text{MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 1.0) (Pt2 1.0 0.0)))})
\]

\[
(\text{MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 0.0) (Pt2 0.0 1.0)))})
\]

\[
(\text{MkFrame (Line2ToVector2 (MkLine2 (Pt2 1.0 1.0) (Pt2 1.0 0.0)))})
\]

**EmptyFrame**

\[
)
\]

\[
)
\]

**Example 2**

The following example illustrates how a picture can be constructed using two simpler frames.

Let \( \text{frame} \) be the one described in example above (Figure 3.10 [a]). If \( \text{addendum} \) is a frame defined as follows (Figure 3.10 [b]),
[a] Frame frame

[b] Frame addendum

[c] Frame (UnionFrame frame addendum)

Figure 3.10 Examples of Frame objects
addendum = (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 1.0) (Pt2 1.0 1.0)))
              EmptyFrame),

then the frame represented by (UnionFrame frame addendum) is shown in Figure 3.10 [c].

Operations on Frames

There are two broad classes of operations on frames: geometric and boolean. Geometric operations include the conventional transformations such as rotation, translation and scaling. These operations are defined on picture modules in classical graphics. Boolean operations include union, intersection and subtraction. While union of picture modules is supported by conventional graphics functional standards (except GKS), the other two boolean operations are generally left to the modeling system designer (who uses the graphics functional standard to build a modeling tool) to be implemented. In FunGraf, we were unable to implement the intersection and difference operations due to certain limitations of the Haskell interpreter.

Geometric Operations

Geometric operations defined on frames include rotation, scaling and translation. Geometric operations on boolean frame expressions are defined as the same boolean expression of the transformed frames. This is to optimize the computation of boolean frame expressions (more on this later).

The function scaleFrame scales the given frames with the origin as the pivot. If a different pivot is desired, the user (programmer) must explicitly translate the pivot to origin, scale the frame and then translate the origin to the pivot (Figure 3.11 [a]).

Translation and rotation are similarly defined (Figure 3.11 [b]).

Boolean Operations

Union of two frames is a frame that includes Vector2s of both the frames. Intersection of two frames is a frame that includes only those points that lie in both the frames. Difference of two frames f1 and f2 is a frame that includes those points of f1 that do not lie in f2.

When a boolean operation is sought to be performed on two frames, a boolean expression is created identifying the operation. Evaluation of these boolean expressions, i.e., the process of computing the result (frame) of a given boolean expression can be expensive. Hence the evaluation of such expressions is deferred as much as feasible. This is achieved by not
demanding the value of the expression unless the expression needs to be displayed. Subjecting a
boolean expression to a geometric operation also does not necessitate the evaluation of the
expression due to the fact that the order of application of boolean and geometric operators can be
reversed. Hence,

\[ <\text{geom-op}> ( <\text{boolean-op}> f1 \ f2 ) = <\text{boolean-op}> ( <\text{geom-op}> f1 \ f2 ) \]

where \(<\text{boolean-op}>\) represents a boolean operation and \(<\text{geom-op}>\) represents a geometric
operation.

The operation that will force the evaluation, however, is the frame display operation. When
a boolean expression is sought to be displayed, its evaluation cannot be deferred any more.

3.5 Extension of Haskell Input-Output System for Graphics

A meaningful graphics system cannot be built without adequate support for input-output
operations. Defining functional graphics I/O has been one of the major tasks in this project. In
the following sections, we describe the standard graphics I/O functionalities that have been
added to the Haskell I/O framework to support interactive graphics. This formal description is
followed by example programs illustrating the usage of this extension. However such a
description assumes, on the part of the reader, an understanding of certain keywords used
commonly in the graphics community.

A brief description of such terms is given in the following section.

3.5.1 Commonly Used Keywords in Graphics

Workstation

A computer system that is equipped with the necessary hardware, software and input-output
devices to run computer graphics application programs [Foley and van Dam 1990].

In FunGraf, the term \textit{workstation} refers to an abstraction comprising three entities: the
window used for display, the input-output devices and the associated software. Hence,
initializing a workstation with \texttt{openWorkStation} creates the X display connection, the display
surface (window) as per the specified attributes and the tools required for software simulation
of some of the input-output devices. Clearly, the same physical machine may be referred to by
different "workstations".
(a)

scaleFrame :: Frame -> Float -> Frame
scaleFrame EmptyFrame _ = EmptyFrame
scaleFrame _ 0.0 = EmptyFrame
scaleFrame (MkFrame v f) s =
    MkFrame (scaleVector2 v s) (scaleFrame f s)
scaleFrame (UnionFrame f1 f2) s =
    UnionFrame (scaleFrame f1 s) (scaleFrame f2 s)
scaleFrame (IntersectFrame f1 f2) s =
    IntersectFrame (scaleFrame f1 s) (scaleFrame f2 s)
scaleFrame (FrameDifference f1 f2) s =
    FrameDifference (scaleFrame f1 s) (scaleFrame f2 s)

(b)

-- Translation of a Frame
moveFrame :: Frame -> Point2 -> Frame
moveFrame EmptyFrame _ = EmptyFrame
moveFrame (MkFrame v f) to =
    Mkframe (moveVector2 v to) (moveFrame f to)
moveFrame (UnionFrame f1 f2) to =
    UnionFrame (moveFrame f1 to) (moveFrame f2 to)
moveFrame (IntersectFrame f1 f2) to =
    IntersectFrame (moveFrame f1 to) (moveFrame f2 to)
moveFrame (FrameDifference f1 f2) to =
    FrameDifference (moveFrame f1 to) (moveFrame f2 to)

Figure 3.11 Operations on the Frame object
(a) Scaling (b) Translation
(c)

-- Rotate the given Frame by the given angle about the given Pivot.
rotateFrame :: Frame -> Float -> Point2 -> Frame
rotateFrame EmptyFrame ___ = EmptyFrame
rotateFrame f 0.0 ___ = f

rotateFrame (MkFrame v f) phi point =
    MkFrame (rotateVector2 v phi point)
        (rotateFrame f phi point)

rotateFrame (UnionFrame f1 f2) phi point =
    UnionFrame (rotateFrame f1 phi point)
        (rotateFrame f2 phi point)

rotateFrame (IntersectFrame f1 f2) phi point =
    IntersectFrame (rotateFrame f1 phi point)
        (rotateFrame f2 phi point)

rotateFrame (Frame Difference f1 f2) phi point =
    Frame Difference (rotateFrame f1 phi point)
        (rotateFrame f2 phi point)

Figure 3.11 Operations on Frame Object (contd)

(c) Rotation
Graphics Functional Standard

A specification of a set of operations for drawing, input-output, workstation management and resource handling. In conforming to this standard, the various workstation vendors adhere to certain specifications in defining the aforesaid operations. The consequence of this is that an application built using the standard is portable across a large number of workstations of different makes.

Display Surface

The extent on the display device used for display operations. In windowing systems, this refers to the particular window under consideration.

World Coordinate System

The coordinate system adopted by the application to define the models of images that it creates, manipulates and finally displays on the workstation.

Device Coordinate System

An integer coordinate system that is mapped on to the display surface. Corresponding to every pixel on the surface, there is a point (a pair of whole numbers). The coordinate system spans a limited extent: the x and y coordinates assume integer values from 0 to a certain maximum. Due to the discrete nature of this coordinate system, some information is lost when images are mapped from the world coordinate system to this system.

Normalized Device Coordinate System (NDC)

A coordinate system that is used to store the images to be mapped onto a device in a device independent form. This coordinate spans a limited extent: the x and y values range from 0.0 to 1.0. The image that is in the NDC has gone through the rest of the viewing pipeline and is ready to be displayed. A point in the NDC image corresponds to a point on the display surface, with X and Y coordinates normalized by the X and Y extents respectively of the display surface.

Window

A section of the image in the world coordinates selected for display. Also used to denote the entity that demarcates such a section. For our purposes, windows will be rectangular with one pair of the sides parallel to each of the axes. In FunGraf, we consider only two dimensional images. In such a system, a window is specified by giving its lower left and the top right vertices.
Viewport

A section of the display surface which is to be used for displaying the window. Viewports may also be defined in NDC. Viewports are always two-dimensional rectangles defined in the coordinate system $0.0 \leq x, y \leq 1.0$. Hence the viewport $< (0.0, 0.0), (0.5, 0.5)>$ selects only the lower half of the display surface for displaying images.

Logical Devices

The definition of an input-output functionality that will be used by the graphics application. There are six such logical devices: Locator, Pick, Valuator, StringInput, Stroke and Choice. These six devices are merely abstractions; physically, such functionality may be implemented by the same device or by different devices. In fact, no physical device may exist to implement a logical device. In such an event, software must simulate the operations of the logical device [Rogers 1991].

Physical Device

The hardware that performs the functions of a logical device.

Locator

The logical input device that returns the point (two dimensional) on the display surface that was clicked upon by the user. The corresponding physical device may be a mouse or a tablet.

In FunGraf, the physical device implementing the locator device is the mouse.

Pick

The logical input device that returns the frame that was clicked upon by the user. Its very nature makes it impossible for this device to be implemented in any way other than by software simulation.

Valuator

The logical input device that allows the user of the application to input to the application a scalar (real) value. The corresponding physical device implementing this functionality may be a joy-disk, potentiometer, joy-stick, etc..

In FunGraf, this device has been simulated using the scrollbar software structure. The scrollbar is operated by the mouse pointer.
Stroke

The logical input device that allows the user to input to the application a sequence of points. In FunGraf, the mouse operation has been redefined so that the mouse itself may be used to implement this device. When the user depresses the mouse button and releases it, the mouse implements the locator device. When the user keeps the button depressed, the mouse inputs a sequence of points to the application. This sequence is terminated when the user releases the button.

StringInput

The logical device that inputs a character string. The keyboard device is used to implement this functionality.

Choice Device

Also referred to as the Button Device, this logical device has the functionality of indicating to the application the user's selection from a number of given options. While a specialized set of function keys may be used in hardware to implement this operation, a more common approach is to simulate this device in software.

3.5.2 A Framework for Graphics Input-Output

The Haskell input-output system has been extended to define and implement the five logical devices defined above. In addition, the display operation (display a frame) too has been defined and implemented in the Haskell system. These operations have been fashioned in the continuation style. It must be noted that while it seems appropriate to think of these different devices as being different I/O channels (just like stderr, stdin and stdout Unix channels), in the current extension all these devices are operated upon using one channel termed the graphics channel. The graphics channel is, appropriately enough, the workstation identification returned by the openWorkStation operation.

In the following paragraphs, we present the features that constitute the I/O extension. In order to better understand the framework to be presented, the reader should become familiar with the preliminary definitions given in Figure 3.1.

Workstation Initialization

All the graphics input-output functions operate on the graphics channel, loosely referred to in this discussion as the workstation. Initializing a workstation involves creating and
initializing the data structures implementing the channel, assigning resource values to the
channel resources, and finally marking the channel identification as a valid channel. The effects
of input and output operations can be altered by the values assigned to the channel attributes. A
limited set of attributes is currently defined (Figure 3.12). This list can be easily extended to
include more complex attributes.

The workstation initialization operation is performed by the openWorkStation function:

\[
\text{openWorkStation} :: \text{String} \rightarrow \text{Attributes} \rightarrow \text{FailCont} \rightarrow \text{WsCont} \rightarrow \text{Dialogue}
\]

The function takes as argument the name of the workstation, the list of attribute-value pairs, the
function to be called in case of error (failure continuation) and the function to be called in case
the call succeeds (success continuation). In the current implementation, the workstation name
happens to be the internet address of the workstation. The Haskell interpreter passes to the
failure continuation the type of the error encountered. The success continuation takes as its first
argument the valid channel identification, which is then used for further operations.

Locator Operation

The locator operation is defined by the readLocator function that scans the specified channel
for a mouse click. It returns to the application the coordinates of the selected point in normalized
form. If the channel is not valid or should any other error occur in scanning for input, the
function invokes the failure continuation with the appropriate error description. When the scan
is successful, the function invokes the success continuation with the input as a Point2 type
argument. Consequently, readLocator has the following type:

\[
\text{readLocator} :: \text{Channel} \rightarrow \text{FailCont} \rightarrow \text{LocCont} \rightarrow \text{Dialogue}
\]

Furthermore, if the user clicks upon a region not lying within the window specified for the
workstation, readLocator returns the null entity, EmptyPoint2.

Pick Operation

The pick operation enables the user of the application to indicate the selection of a picture
module to the application. The operation is defined to return the smallest enclosing picture
Window
Viewport
WorkStation_Width
WorkStation_Height
ValuatorNeeded
LineType
MarkerType

Figure 3.12: Channel Attributes
module (frame or scene) that the selected point is contained in. Also, if the selected point lies outside the workstation window, then pick operation returns the null entity (EmptyFrame or EmptyScene). In 2D graphics, pick handles selection of frames; in 3D graphics, it handles that of scenes. In the current system, however, the pick operation has been defined to handle frames alone. Using this operation, the user of the application may point out to the application the identity of a frame that has been displayed on the workstation. This logical device has been implemented using software simulation. The function defining this logical device operation, namely pickDevice, has the following type:

\[
pickFrame :: \text{Channel} \to \text{Window2} \to \text{Window2} \to [\text{Frame}] \to \text{FailCont} \to \text{LocCont} \to \\
\text{Dialogue}
\]

The pickFrame function invokes the readLocator function. The Point2 type object returned by the latter function is then checked to see if it lies within the specified window (the second argument to the function; type Window2). If so, the point is checked to see if it belongs to any of the frame objects passed in the fourth parameter. If \( p \) is the point returned by the readLocator function (type Point2), then for every frame \( f \), the function performs the following action:

\[
\{ \text{Let } p \text{ be the point (type Point2) returned by readLocator call } \} \\
* \text{ if } f \text{ is EmptyFrame, then return } f \\
* \text{ if } f \text{ is of the form (MkFrame vector frame), then} \\
  - \text{ if } p \text{ is the vector (type Vector2) vector then return frame } f \\
  \quad \text{else} \\
  \quad \text{if Point2 } p \text{ belongs to frame frame then} \\
  \quad \quad \text{if frame is in normal form then return frame } f \\
  \quad \quad \text{else} \\
  \quad \quad \quad \text{return the sub-frame of frame frame to which Point2 } p \text{ belongs} \\
  \quad \quad \quad \text{(could be frame itself)} \\
  \quad \text{else return EmptyFrame} \\
\]

* \text{ if } f \text{ is of the form (UnionFrame } f1 \ f2 \text{) then} \\
  - \text{ if Point2 } p \text{ belongs to frame } f1 \text{ then} \\
  \quad \text{return the sub-frame of } f1 \text{ to which } p \text{ belongs (could be frame itself)} \\
  \quad \text{else} \\
\]
if Point2 p belongs to frame f2 then
    invoke the function with frame f2 and Point2 p

* if f is of the form (IntersectFrame f1 f2) then
    - if Point2 p belongs to f1 then
      if Point2 p belongs to frame f2 then
        return (IntersectFrame f1 f2)
      else
        return EmptyFrame
    return EmptyFrame

* if f is of the form (FrameDifference f1 f2) then
    - if Point2 p belongs to frame f2 then
      return EmptyFrame
    else
      invoke the function with frame f1 and Point2 p

Valuator Operation

The valuator device is an input device using which the user may input to the application a scalar (real) value. The value returned by the valuator lies between 0.0 and 1.0. It may be projected by the application to make it lie in any interval on the real line. While the physical device implementing the valuator functionality is often a potentiometer based device (such as a joy-disc), many applications simulate valuator device using software.

In FunGraf, we have simulated the valuator device using an X scrollbar. The function performing this input is readValuator and is of the type

\[
\text{readValuator} :: \text{Channel} \rightarrow \text{FailCont} \rightarrow \text{ValCont} \rightarrow \text{Dialogue}
\]

ValCont is the type of a function that takes a float argument and returns Dialogue. Clearly, the success continuation of readValuator decides the further course of the program by processing the value returned by the successful valuator input. The failure continuation may be invoked if the channel is not valid or if the channel is not equipped with a valuator.
Stroke Device Operation

A stroke device may be used to input to the application a sequence of points. The most common use of this logical device is in tracing the path of the locator device. The system begins tracking the path position of the locator position in response to a certain user action. Thus the application receives a steady stream of Point2 objects. The system aborts the tracking in response to another action of the user. Consequently, two actions are defined in the operation of this device: one that begins the input of the list of Point2s and the another that signals the termination of this list. The semantics of the function defining this device operation are such that the function causes its success continuation to be invoked when the stroke device returns a valid object of type Point2. When the user terminates the input, the device returns an error. In this event, the function readStroke invokes the specified failure continuation. Consequently, while the device inputs a steady stream of points, one call to readStroke yields only the next element in this list. The application must be so structured that the success continuation processes the input of the next point and issues a recursive call to readStroke, thereby completing the input loop. The failure continuation must be defined to handle the termination of the input list. Hence, the function readLocator itself can be used to implement this logical device operation:

\[
\text{readStroke} :: \text{Channel} \rightarrow \text{FailCont} \rightarrow \text{LocCont} \rightarrow \text{Dialogue}
\]

For similar reasons, the physical device that implements the locator operation can also be made to implement the stroke operation with a small difference in the semantics of the operation.

String Input Device Operation

This device has a functionality similar to the keyboard. The input function is

\[
\text{readStrInputDevice} :: \text{Channel} \rightarrow \text{FailCont} \rightarrow \text{StrCont} \rightarrow \text{Dialogue}.
\]

The success continuation is a function that takes a string argument and continues execution of the program.

Picture Module Display Operation

This is by far the most important output operation. The data model of the application is visualized in terms of the data types of the FunGraf system, defined as picture modules. The picture modules are in turn manipulated according to the user's commands using the input
devices, projected into a frame. The frame so obtained is finally viewed on the display device using the display function. The following discussion will be restricted to the display of two dimensional picture modules, namely frames.

The type of the frame display function is

\[
\text{showFrame} :: \text{Channel} \to \text{Frame} \to \text{Window2} \to \text{Window2} \to \text{FailCont} \to \text{SuccCont} \to \\
\to \text{Dialogue}
\]

The function also takes the specified window (type Window2) and a viewport (type Window2), along with the frame that is to be displayed. As mentioned earlier, the window argument identifies the section of the application model that is currently of interest. The viewport serves to identify the section of the display surface upon which the frame is to be displayed. The showFrame operation proceeds as follows:

Step 1: Convert the frame to its normal form.

* If the frame is of the form

\[(\text{UnionFrame } f_1 f_2 )\]

then invoke showFrame to display \( f_1 \) and \( f_2 \) sequentially. The output of \( f_2 \) will be the success continuation of the output of \( f_1 \).

* If the frame is of the form

\[(\text{IntersectFrame } f_1 f_2 ) \text{ or } (\text{FrameDifference } f_1 f_2 )\]

then the frame cannot be displayed. This is because, owing to certain limitations, the frame normalization operation could not be implemented in the system. Of course, the implementation of this normalization function would have meant the complete implementation of the boolean operations. As mentioned earlier in this report, this is a limitation of the implementation.

Step 2: If the frame is \textit{EmptyFrame}, then invoke the success continuation.

Step 3: If the frame is of the form

\[(\text{MkFrame } \text{vector } \text{frame})\]

then invoke the vector output operation
displayVector2 :: Channel -> Vector2 -> Window2 -> Window -> FailCont ->
                -> SuccCont -> Dialogue

to display the Vector2 object vector. If this succeeds, then invoke showFrame as
success continuation to display the rest of the original frame, namely frame. Should
there be an error in displaying the vector, invoke the failure continuation.

It is worth noting that the functionality of displayVector2 includes the clipping of the
Vector2 object with respect to the window passed to it as its argument. However, in the current
implementation, displayVector2 does not perform this clipping operation. The projection of the
Vector2 object to make it lie on the display surface indicated by the viewport argument is done
by the lower level T and C routines that actually interact with the underlying X Window system
and draw the Vector2 object on the workstation.

Simulation of Choice Device

The Choice device or the Button device is a logical device that is used to convey to the
application the user’s selection of an option from a given number of options. The application built
using the FunGraf system has amply demonstrated that this device can be simulated using the
pickFrame operation.

3.6 Illustration of Common User Interface Tasks Using the
FunGraf Primitives

The concluding section of this chapter is devoted to illustrating the use of the framework
described in the preceding sections in implementing some common user interface tasks.

Menu Button Implementation

Menu buttons are the most commonly used mechanism to implement the choice functionality
in the modern Graphical User Interfaces. This mechanism can be implemented using the
pickFrame input operation quite elegantly. Suppose, for instance, that the three choices save,
abandon and cancel need to be provided. The following method may be used:

Step 1: Define

\[
\text{Frame } \text{save} = \text{MkFrame} \ (\text{LabelToVector} \ \text{save}-\text{label}) \quad \text{EmptyFrame}
\]
Frame abandon = MkFrame (LabelToVector abandon-label) EmptyFrame
Frame cancel = MkFrame (LabelToVector cancel-label) EmptyFrame

where save-label, abandon-label and cancel-label are objects of type Label that contain the strings 'save', 'abandon' and 'cancel' respectively. The tag LabelToVector merely serves to convert the type of the object from Label type to Vector2 type.

Step 2: The pickFrame operation may be used as follows to identify user selection:

\[
pickFrame \quad channel \quad [ \text{save, cancel, abandon} ] \quad \text{window viewport}
\]
\[
\quad \text{errorHandler}
\]
\[
\quad \text{processChoice}
\]

where the function processChoice, which is the success continuation of the pick call is a function that takes as argument a frame corresponding to one of the labels and processes this choice.

Step 3: The function processChoice may be defined as follows:

\[
\text{case choice of}
\]
\[
\quad 1 \rightarrow ( \text{perform the save operation})
\]
\[
\quad 2 \rightarrow ( \text{perform the abandon operation})
\]
\[
\quad 3 \rightarrow ( \text{perform the cancel operation})
\]
\[
\quad 0 \rightarrow ( \text{illegal or no choice ; indicate error message})
\]
\[
\text{where choice} = \text{frameToActionCode selected-frame}
\]

Note that this function takes as argument the selected frame, selected-frame. It is easy to visualize a function that returns the appropriate integer depending on the selected frame object.

Repeatepcnt Execution of Actions
User interfaces rely on being able to repeatedly execute a set of actions until a termination condition is met. This is in general true of graphics applications that provide for interaction
with the user.

To illustrate how this may be elegantly implemented using continuations and (tail) recursion, consider a simple user interface that displays on the screen three options, ADD, DELETE and EXIT. When one of the first two options is selected by the user using the locator device, the name of the option is printed on the console and the operation repeats. When the user selects the EXIT option, the program aborts execution. A more useful program would, of course, manipulate the data structures of the applications appropriate to the action and probably invoke nested interfaces in response to certain actions. However, the purpose here is merely to demonstrate the methodology of going about defining such interfaces. For the sake of simplicity certain details such as opening of the workstation and composing the menu "buttons" have not been shown.

Consider the functionality to be implemented by a function menu that takes as argument a valid channel and a frame containing the menu button descriptions. Since the function performs input-output, it returns the type Dialogue. Thus, the type of the function is:

\[
\text{menu :: Channel -> Frame -> Dialogue}
\]

The function performs the following steps:

Step 1: Display the menu options on the screen. Since the second argument contains the definition of these buttons, it must be displayed using the function:

\[
\text{showFrame channel frame window viewport errorHandler success-continuation}
\]

The identifiers window, viewport and errorHandler are assumed to be globally known. Window is the window that identifies the section of the 'world' to be displayed. Viewport is the viewport of the workstation and errorHandler is the function that handles the errors that may be encountered in displaying the frame.

Step 2: After displaying the menu successfully, the program must identify the user's selection of one of the menu objects. This is achieved by performing the pick operation with the list of the menu buttons to select from.

\[
\text{pickFrame channel window viewport [frame] errorHandler success-continuation}
\]
The object frame is a list of menu buttons, each of which is in turn a frame. Since this action is performed only if step 1 is successful, this will be the last success continuation of the display operation of step 1.

Step 3: The object selected by the user (a frame), is converted to a number by a simple function. Assume that the numbers 1, 2 and 3 are respectively assigned to options ADD, DELETE and EXIT. Assume also that the number zero is assigned to any selection that does not correspond to these options (illegal selection).

Step 4: As we saw in the previous section a simple "case" statement decides on the action to be performed depending on user selection (i.e., the number obtained in step 3). This construct is as follows:

\[
\text{case choice of}
\]
\[
0 \rightarrow \{ \text{Illegal choice : take no action and recurse } \}
\]
\[
\text{menu channel frame}
\]
\[
1 \rightarrow \{ \text{Option "ADD" : print "ADD" and recurse } \}
\]
\[
\text{appendChan stdout "ADD" errorHandler (menu channel frame)}
\]
\[
2 \rightarrow \{ \text{Option "DELETE" : print "DELETE and recurse } \}
\]
\[
\text{appendChan stdout "DELETE" errorHandler (menu channel frame)}
\]
\[
3 \rightarrow \{ \text{Option "EXIT" : abort execution } \}
\]
\[
\text{done}
\]

A more practical program would clean up the data structures and close the workstation prior to aborting execution.

The complete listing of this program is given in Figure 3.13. Note that the efficiency of this program depends a great deal on the efficient implementation of recursion by the compiler.

**Tracking the locator cursor**

A more rigorous user interaction is the tracking by the application of the locator (mouse) cursor. This operation is used by many applications to implement the "paint" operation where the user is allowed to use the mouse cursor as a "paint brush" to paint the screen with. In this section we illustrate how this may be implemented using the FunGraf framework. Assume that
module Menu where

main :: Dialogue

main = openWorkStation workstationName [] errorHandler (channel -> menu channel panel)

menu :: Channel -> Frame -> Dialogue

menu channel panel = showFrame channel panel window viewport [] errorHandler

(pickFrame channel window viewport [panel] errorHandler

(selected_frame ->

  case command of
    0 -> menu channel panel
    1 -> appendChan stdout "ADD" abort (menu channel panel)
    2 -> appendChan stdout "DELETE" abort (menu channel panel)
    3 -> appendChan stdout "EXIT" abort done

  )

)

button_to_command :: String -> Int

button_to_command "ADD" = 1
button_to_command "DELETE" = 2
button_to_command "EXIT" = 3
button_to_command _ = 0

label_string :: Frame -> String

label_string (MkFrame (LabelToVector2 (MkLabel string _ _)) _) = string

label_string _ = "None"

window , viewport :: Window2

window = MkWindow2 (Pt2 0.0 0.0) (Pt2 100.0 100.0)
viewport = MkWindow2 (Pt2 0.0 0.0) (Pt2 1.0 1.0)

panel, add, delete, exit :: Frame

add = MkFrame (LabelToVector2 (MkLabel "ADD" (Pt2 10.0 3.0) window viewport)
delete = MkFrame (LabelToVector2 (MkLabel "DELETE" (Pt2 30.0 3.0) window viewport)
exit = MkFrame (LabelToVector2 (MkLabel "EXIT" (Pt2 60.0 3.0) window viewport)

Figure 3.13 Repetition of actions using continuation I/O and recursion
the function \textit{track} implements this functionality and that function \textit{track} takes as argument the valid channel obtained by opening the workstation. The type of \textit{track} is hence

\textit{track} :: Channel \rightarrow Dialogue

The task is performed in the following steps:

Step 1: Since a sequence of points is expected to be input, the first step is to read the stroke device using the call

\textit{readStroke} channel failure-continuation success-continuation

As was observed in the description of the stroke device earlier, the success continuation is invoked when another valid point is supplied by the stroke device to the application. When the user performs the action to terminate the input list, the failure continuation will be invoked. Hence the failure continuation merely causes the program to exit.

\textit{readStroke} channel

\hspace{1em} \textit{exit} \hspace{1em} \text{-- termination of input list; abort execution}

\hspace{1em} \textit{success-continuation} \hspace{1em} \text{-- one more point read in; process the point!}

The success continuation is more complex. This function is described in the following steps.

Step 2: The success continuation takes as argument the point (type \textit{Point2}) supplied by the stroke device, plots it on the display device and invokes \textit{track} recursively. Prior to being plotted, however, the point must be transformed from the normalized device coordinates to the "world" coordinates. The reader would remember that the locator and stroke devices return the selected point in normalized device coordinates. Transforming coordinates from normalized device coordinates to the window space is done by the function

\textit{coordXform} :: \textit{Point2} \rightarrow \textit{Window2} \rightarrow \textit{Window2} \rightarrow \textit{Point2}
The first argument is the point to be transformed. The second and third arguments represent the "source" window (viewport, in this case) and the "destination" window (application window).

Step 3: After the point that is to be plotted has been transformed, it must be encapsulated in a frame before it can be displayed. This is done as follows:

\[
frame = MkFrame \ (Pt2ToVector2 \ point) \ EmptyFrame
\]

where \( <point> \) is the point computed in step 2. \( Pt2ToVector2 \) is merely a tag to type cast the Point2 object to a more general Vector2 object type. Note that the use of \( EmptyFrame \) is to comply with the definition of frame object.

Step 4: Now that the frame containing the point has been computed, it may be displayed using the \( showFrame \) operation.

\[
showFrame \ channel \ window \ viewport \ frame \ errorHandler \ success-continuation
\]

Function \( errorHandler \) handles errors encountered in I/O operations. If this operation is performed successfully, the program would have completed one cycle of operations successfully and must now recurse. Hence the success continuation of this operation is a recursive call to the \( track \) function.

\[
showFrame \ channel \ window \ viewport \ frame \ errorHandler \ ( \ track \ channel \ )
\]

The complete listing of this program can be found in Appendix B.

These two examples conclude this section. These examples merely serve to illustrate how the FunGraf framework may be used to build practical applications. More complex programs illustrating the practicability of the system have been included in Appendix B.
Chapter IV

Notes on Implementation

4.1 Purpose of this Chapter

This chapter describes the implementation details of the FunGraf system. Also described are certain problems that were encountered during implementation. The aim of recording the implementation details is mainly to assist in the evaluation of the performance of the system. These details would also help anyone attempting to further enhance the system. The structure of this chapter is described below.

First, we present the implementation environment and the layer structure of the system. In the subsequent sections, we describe the important implementation issues in interfacing adjacent layers such as the mechanism of data exchange across interfaces. Since Haskell lists are delayed and T lists are not, we describe the mechanism of passing lists across the Haskell-T interface. The most important implementation issue in the T-C interface is the mechanism of invoking C functions in T environment. We have implemented the workstation and graphics related data structures in C. In the description of this layer, we present the details of these data structures. In the concluding sections, we discuss in detail the procedures implementing each Haskell graphics I/O operation.

4.2 General View of the System Structure

Hardware Platform

The FunGraf system has been implemented on DECStation 5000 running Ultrix version 4.2, Rev 96. DECStation 5000 uses a MIPS 3000 RISC processor.

Software Environment

A pictorial representation of the system structure is given in Figure 4.1. FunGraf has been built wholly using the features provided by the Yale Haskell implementation. The current version of Yale Haskell is implemented in T, a dialect of LISP. Haskell functions are translated into T functions which are subsequently interpreted. The T functions that implement the standard
Figure 4.1: Layer Structure
Haskell I/O operations interact directly with the Ultron operating system.

In extending the Haskell I/O system to include graphics operations, we implemented a new set of T primitives and interfaced them to a set of C functions which provide the basic workstation management and graphics capabilities to the system. The C functions perform miscellaneous operations such as providing basic workstation management operations, computing the data required to perform graphics output or to read input and issuing appropriate Xlib calls to perform the relevant operations.

Hence, there are three interfaces involved: the Haskell-T interface, the T-C interface and the C-Xlib interface. Of the three, the last interface is a standard that has merely been used in this project. The following paragraphs describe the first two interfaces.

**The Haskell-T Interface**

Yale Haskell establishes a simple mechanism to interface T and Haskell functions. The purpose of this interface is to enable the invocation of a T function by a Haskell function, to translate appropriately the Haskell parameters to T types, and finally to reconvert the value computed by the invoked T function to the appropriate Haskell type prior to returning the value. When Haskell invokes a T function, the latter function is referred to as a "primitive". The interface defining function `define-primitive` defines the following attributes, among many others, of the primitive:

a) the **arity** of the primitive, that is, the number of parameters that the primitive would be invoked with.

b) the 'Haskell type' of the primitive, i.e., the type of the primitive as it is seen in Haskell. This is defined as a string of characters enclosed in quotes.

c) the T function which will be invoked whenever the primitive is invoked from a Haskell function.

Once this binding has been compiled and the different modules linked together, the interface becomes complete. For instance, consider function \( X \) with the type signature given below.

\[
X :: A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \ldots \rightarrow A_{n-1} \rightarrow A_n
\]

\( X \) is established as a primitive by the definition:
(define-primitive "X" ((arity n-1) (type "A_1 -> A_2 -> A_3 -> ... -> A_{n-1} -> A_n"))

(lambda (V_1 V_2 V_3 ... V_{n-1}) ...)
)

The body of the lambda block is the set of T statements to be executed every time the
primitive X is invoked. V_1, V_2, ..., V_{n-1} are the formal parameters of types A_1, A_2, A_3, ..., A_{n-1}
respectively, corresponding to the (n - 1) actual parameters of the primitive.

Parameter passing deserves special attention. Not all Haskell types may be passed down to T.
Particularly important are lists as parameters to primitives. This is so because while Haskell
lists are delayed, i.e., the tail of the list is unevaluated while the head of the list is demanded, T
lists have no such property. The difference between Haskell and T list structures is pictorially
shown in Figure 4.2 [a] and [b].

When a Haskell list is passed to a T function, the list itself is not delayed but its elements are.
The list is of the form (car . cdr), where each of car and cdr is a 'delay' (Figure 4.2 [a]). In
turn, each of the head and tail is tagged with a boolean value indicating if the element has been
evaluated (True) or not (False). To get a T list (Figure 4.2 [b]) from a Haskell list, the function
hlist->tlist (Figure 4.3 [a]) is used which forces the evaluation of car and recursively forces
that of cdr. In converting a Haskell list to a T list, each element must be evaluated (i.e., the tags
associated with each element must be set to True). This is done by the function hforce. The
function tlist->hlist (Figure 4.3 [b]) performs the reverse operation and is used when a
computed T list has to be returned to the primitive as a Haskell list. The function hdelay
converts the ordinary list into a lazy list by traversing down the list recursively and 'delaying'
each element (setting the tag associated with each element to False).

The T-C Interface

Since the underlying platform that provides to the system its basic graphics capabilities
is the X Window System and since (at the time of writing) only C language bindings to X functions
exist (the Xlib-C binding), and further since the Haskell system's interface to C is through T,
the T-C interface becomes quite significant. C language functions may be invoked from the T
environment using the define-foreign mechanism of T. This mechanism essentially binds a T
function declaration to a C language function so that invoking the T function causes the actual
parameters to be transferred to the C function, the latter function to be executed and the return
[a] A T List

(1 2 3)

[b] Equivalent Haskell (delayed) List

(((delay . ( l () 1 )) .
(delay .
  ( l () (( delay . ( l () 2 )) . (delay
    . ( l () (( delay . ( l () 3 )) . (delay
      . ( l () (())))))))
))

[c] After Force Evaluating the Whole List

(((nil . 1) . (nil . ((nil . 2) . (nil .
  ((nil . 3) . nil))))))

Figure 4.2 : List Structures in Haskell and T
[a] hlist→tlist : convert a Haskell list to a T list

(define (hlist→tlist list)
    (if (null? list) ;; Is the list empty?
        nil ;; If so, return nil
        (cons (hforce (car list))
            ;; Evaluate first element
            (hlist→tlist (hforce (cdr list))
                ;; then evaluate rest of the list
            )
        )
    )
)
)
)

[b] tlist→hlist : convert a T List to a Haskell list

(define (tlist→hlist list)
    (cond ((null? list) nil) ;; Is the list empty?
        (else
            ;; "delay" the first element
            (cons (hdelay (car list))
                ;; then "delay" the rest of the
                ;; list
                (hdelay (tlist→hlist (cdr list)))
            )
        )
    )
)
)
)

Figure 4.3 : Interconversion of T and Haskell List Structures
value of the function to be reflected as that of the T function. Hence, the following pieces of information go into this binding:

1. the name of the T function
2. the name of the C function
3. for every parameter passed to the C function:
   (i) the name of the parameter
   (ii) the nature of the parameter (input to the called module, output from the called module or both)
   (iii) the type of the parameter (character, integer, string, etc)
4. type of the value returned by the C function (this may take the value \textit{ignore}, equivalent to the \textit{void} type in C).

The object files formed by compiling this information and those formed by compiling the C modules must be linked together in order to make the binding effective. An example illustrating such a binding is shown in Figure 4.4.

At this writing, parameters of type \texttt{Float} cannot be passed across the T-C interface. Consequently, we adopted a work-around by which floating point numbers that are required to be passed across the T-C interface are converted to a character string by the calling function and are converted back to floating point type by the invoked function. This is significant because of the memory required to store the intermediate character string is allocated in the heap. Since some high precision floating point numbers may require a large amount of space to store a large mantissa, space for an arbitrarily large character string must be allocated. Graphics operations rely on extensive floating point arithmetic, particularly the image display operation \textit{showFrame}, which for a reasonably complex picture module might cause thousands of floating point numbers to be passed across the T-C interface. Thus, the execution efficiency of the graphics applications built using this framework depends to a great extent on the garbage collection operation of T runtime system. This point must be taken into account while evaluating the efficiency of this framework.

4.3 Implementation of Graphics Input-Output Framework

The most significant aspect of this project has been extending the Haskell I/O system to include graphics capabilities. We consider it important to document how this extension has been implemented so that it may be enhanced in the future. The following sections describe the implementation of the graphics devices and the corresponding I/O operations. Each of the
[a] Defining the Interface in T

(define-foreign OpenDisplay ;; "OpenDisplay" is a
   ;; foreign function
   ("OpenDisplay"
      (in rep/string) ;; A C/Pascal function
      )
   rep/integer) ;; One value parameter

;; Returns an integer

[b] The C Language Function

int
OpenDisplay ( char *display_name )

   /* One Argument */
{
   /* Xlib calls to open the display named
   * display_name and return an identification. */
}

Figure 4.4: Establishing T-C Interface
[a] Request-Response Mechanism

Haskell Application \[\rightarrow\] Haskell I/O System \[\rightarrow\] T Environment \[\rightarrow\] C Env

Request \[\rightarrow\] Invoke T primitive \[\rightarrow\] \[\rightarrow\] Invoke C primitive \[\rightarrow\] Result

Response \[\rightarrow\] \[\rightarrow\] Packaged Result

[b] List of Graphics Requests and Responses

<table>
<thead>
<tr>
<th>Requests</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenWS</td>
<td>Success</td>
</tr>
<tr>
<td>CloseWS</td>
<td>Failure IOError</td>
</tr>
<tr>
<td>DumpFrame</td>
<td>Str String</td>
</tr>
<tr>
<td>ReadLocator</td>
<td>Loc Point2</td>
</tr>
<tr>
<td>PickFrame</td>
<td>Fr Frame</td>
</tr>
<tr>
<td>ReadValuator</td>
<td>Scalar Float</td>
</tr>
<tr>
<td>ReadStrDevice</td>
<td>Ws Channel</td>
</tr>
<tr>
<td>ReadStroke</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.5 Implementation of Graphics I/O Calls
following sections is concerned with a particular logical device operation and deals with how the device has been implemented (that is, whether the device has been simulated in software or relies on a hardware device) and how the input or output operations performed on the device by the Haskell system is implemented using the X Window system.

The mechanism by which the graphics I/O calls are translated into requests to the Haskell I/O system, the requests translated into calls to T and C primitives, and the results of execution of the T and C primitives converted back to the appropriate response and sent to the calling Haskell application are outlined in Figure 4.5.

### 4.3.1 Workstation Related Operations

Haskell views the workstation as an abstraction with the following components:

1. an identification number (type Channel)
2. a locator device
3. a pick device
4. a valuator device (optional)
5. a keyboard device
6. attributes (Figure 3.12)

The C modules create and maintain data structures that support this abstraction. In particular, a workstation is a C data structure shown in Figure 4.6 [a]. The C modules maintain an array of such workstation objects (Figure 4.6 [b]). Consequently there cannot be more than a maximum number of workstations (MAX_WORKSTATIONS) open at one time. This is not a serious problem since most applications do not open more than a few workstations at any given time.

The Haskell function to open a specified workstation, namely

\[ \text{openWorkStation} :: \text{String} \rightarrow \text{BundledAttributes} \rightarrow \text{FailCont} \rightarrow \text{OpenCont} \rightarrow \text{Dialogue} \]

issues the request

\[ (\text{OpenWS} \ \text{workstation-name} \ \text{attribute-value-list}) \]

to the Haskell I/O system. \text{workstation-name} is the character string identifying the
[a] Workstation Object

```c
struct workstation_type {
    Display *display;  /* X Display connexion */
    Window window;    /* Display Window */
    GC gc;            /* X Graphics Context */
    Valuator valuator; /* Valuator Window Id */
    int thumb_position; /* Valuator thumb
                        * position */
};
```

[b] Workstation Table

![Diagram showing the implementation of the Workstation Object]

`Figure 4.6 Implementation of Workstation Object`
workstation, passed as the first argument to `openWorkStation` and `attribute-value-list` is a list of workstation attribute-value pairs, also passed as argument to `openWorkStation`. In response to this request, the Haskell I/O system invokes the corresponding T function `t_openWorkStation`, which operates in three steps:

1. `t_openWorkStation` first checks to see if the workstation named by the first argument (type String) can be opened. This is done by the underlying C function `CheckIfDisplayExists`, which simply attempts to open the named workstation as an X display.

2. If the workstation can be opened, `t_openWorkStation` invokes the function `t_setWsAttributes` that takes the list of attribute-value pairs (type `BundledAttributes`) and sets the value of each attribute sequentially. The effect of setting the value of an attribute is no more than setting a flag on or off using the underlying C modules. These flags are processed when the C modules initialize the workstation.

3. After all the attribute values have been set, `t_openWorkStation` invokes the C module to open the specified workstation. The C function, `OpenWorkStation`, performs the following steps:

   a) Checks to see if a free slot is available in the workstation array. If not, it returns error.
   b) Opens the X display
   c) Creates an X Window of the dimensions specified by the attributes.
   d) Creates a Graphics Context (an X object) is created and the attributes set using the type `BundledAttributes`.
   e) If the valuator needs to be created (indicated by one of the attribute values), creates an X window to be used to simulate the valuator operation.
   f) Enters all the created elements (step a) through d)) in the next available workstation slot in the workstation array.
   g) If all the steps are successful, returns the position of the newly created workstation object in the workstation array as the channel id and updates the count of workstation slots. In this event, the `openWorkStation` Haskell function takes the success continuation with this channel id as the argument. If any of the steps fail, the function returns with an error condition which in turn causes the
**openWorkStation** function to invoke the failure continuation.

When the program has no more operations to perform on the opened workstation (typically prior to termination), the workstation must be closed. This frees the underlying C data structures that make up the workstation. The function

```
closeWorkStation :: Channel -> FailCont -> SuccessCont -> Dialogue
```

invokes the C function *CloseWorkStation* which in turn closes the X display, destroys the valuator window (if the valuator resource is available), frees the workstation slot in the workstation array and updates the workstation count.

### 4.3.2 Locator Related Operations

This section describes the implementation of the logical locator device and the related operations. The physical device most popularly used to implement the locator functionality is the mouse. Since the platform underlying FunGraf provides X Window mouse as a standard device, it was a natural choice for us to use this physical device.

The main operation defined on the locator is the input operation:

```
readLocator :: Channel -> FailCont -> LocatorCont -> Dialogue
```

*pickFrame*, which relies on mouse operation, is based on *readLocator* function. A call to *readLocator* causes the request

```
(ReadLocator channel)
```

to be issued to the Haskell I/O system, which, in response to this request, performs the following operations:

**Step 1: Check if the locator device exists**

Since every opened workstation is equipped with a locator, this check is essentially to see if *channel* points to a legally opened workstation object in the workstation table. This is done by the T primitive _`t_locatorExists`_ with *channel* as its integer argument. It in turn calls the C function *LocatorExists*. The latter function checks if
channel is a legal index into the workstation table and if the entry pointed to it in the
table contains an opened workstation. Should these checks yield that channel does not
point to a valid workstation, t_locatorExists would return False to the i/O system
which in turn would result in the failure continuation of the readLocator to be
invoked.

Step 2: Read a point from the locator device

The Haskell i/O system calls the T primitive t_readLocator to read a point from the
workstation. The primitive in turn calls C function ReadLocator which interacts with
the X display to read the next mouse click on the window representing the
workstation. ReadLocator issues the standard Xlib calls to wait for the user to press
and release the first mouse button and to obtain the window coordinates of the point
in the workstation window which was clicked upon. Note that this point is an integer
pair and represents a position in the window coordinates, which is the integer
coordinate system with origin at the top left corner. This point is normalized with
respect to the workstation window dimensions (effectively, the window dimensions).
Hence if the mouse click was obtained at point (X, Y) then the point returned to the
calling T primitive is (X/Workstation_Width, Y/Workstation_Height). This pair of
real numbers is returned to the calling T primitive in string form. t_readLocator
in turn converts this pair to a list and then to a delayed Haskell list (using t
list->hlist) and returns to the Haskell i/O system.

Since this is a 'success' response, the Haskell i/O system converts the list into a
Point2 object and invokes the success continuation of readLocator with the response
(LocPr (Pt2 x y)) where (Pt2 x y) is the point read in from the locator.

Note that the application is returned a point whose coordinates are normalized with respect
to the workstation dimensions. It is up to the application to check if this point is in the current
viewport and to map the point from the viewport to the world coordinate space.

Another input function that uses the locator device is pickFrame, which returns to the
application the identity of the frame object clicked upon by the user of the application. As
discussed in the last chapter, pickFrame invokes readLocator to read the point clicked upon by
the user. It then checks the point so obtained to see if it belongs to any frame object passed to it
as a parameter. After the point has been read in using readLocator, the rest of the pickFrame
task is performed entirely in the Haskell domain. The algorithm used by `pickFrame` for membership checking has been described in the definition of the pick logical device (section 3.7.3).

### 4.3.3 Implementation of the Stroke Device

Stroke is a device that is used to input to the application a sequence of points. This device is usually implemented using the tablet, but it can be implemented by the mouse as well. The interpretation of the mouse button clicks have been appropriately defined to view the mouse input as a stroke input or as a locator input. As described in the previous section, the locator read operation waits for the user to press and release the first mouse button at a point without dragging the mouse. The stroke read operation, however, proceeds as follows:

* wait for the user to press the first mouse button
* until the button is released
  * wait for the mouse to be dragged
  * return to the calling application the Point2 operation representing the point where mouse motion was detected
* when the first mouse button is released, return a 'failure' response to the application.

Thus both success and failure responses are used in reading the stroke device. The input interface to the stroke device is the function

\[ \text{readStroke :: Channel} \to \text{FailCont} \to \text{LocCont} \to \text{Dialogue} \]

and its success continuation returns the next point read in the sequence. The success continuation must invoke `readStroke` again to read the next point in the sequence. Hence successive calls to the function read successive points in the stroke sequence. The sequence is terminated by the failure continuation when the button release event is detected.

The call to `readStroke` causes the request `(ReadStroke channel)` to be issued to the Haskell I/O system. Upon receiving this request, the I/O system performs a sequence of operations similar to that performed when the `ReadLocator` request is received. The latter sequence was described in the last section.
4.3.4 Valuator Related Operations

Although devices such as control dial and joy-disk are commonly used to implement the logical valuator device, we have chosen to simulate this device in software. This was described in the previous chapter.

The Haskell I/O function

\[ \text{readValuator} :: \text{Channel} \to \text{FailCont} \to \text{ValuatorCont} \to \text{Dialogue} \]

reads data from the valuator device. If the read succeeds, the success continuation function is called with the input floating point value as its argument. However, should the read fail for any reason, the failure continuation function is invoked. The function readValuator issues the request packet

\[ (\text{ReadValuator} \ channel) \]

to the Haskell I/O system. In response to this request, the following steps are performed by the I/O system.

Step 1: Call \text{t_valuatorExists} to validate the device identification

The purpose of this is twofold: (1) to check if channel points to a legally initialized workstation object and (2) to check if the workstation pointed to by channel has a valuator device. Note that valuator is an optional resource of the workstation and the application may have chosen to open the workstation without a valuator device (ValuatorNeeded = False). The T primitive \text{t_valuatorExists} in turn invokes the corresponding C function ValuatorExists to perform this function. The latter function checks to see if channel is an index to a valid workstation in the workstation array. If so, it further checks to see if the specified workstation has a valid X window id for the valuator device. Should either of these checks yield the value False, the function returns False to the calling T primitive, which in turn returns the failure response packet (Failure <error-message>) to the I/O system, where <error-message> is the error message announcing the failure of the read access. This causes the I/O system to invoke the failure continuation function of readValuator with <error-message> as its argument.
Step 2: Call \texttt{t\_readValuator} to read a value from the device

The T primitive \texttt{t\_readValuator} is called with the integer argument \texttt{channel}. \texttt{t\_readValuator} in turn invokes the corresponding C function \texttt{ReadValuator} with \texttt{channel} as its argument. Since \texttt{ReadValuator} is expected to pass back a floating point value and since the version of T (version 3.1) that was used to implement this system does not support passing of floating point numbers from T to C functions \texttt{t\_readValuator} also allocates a buffer of large size and passes it as the second argument to \texttt{ReadValuator}. \texttt{ReadValuator} obtains the X display connection identification and the X resource id of the X window being used to simulate the valuator function from the workstation object pointed to by the integer index \texttt{channel}. Once these two X resources are known, \texttt{ReadValuator} issues Xlib commands on the X display connection to wait for a mouse click event on the valuator window. As the user clicks on the valuator window, the cursor is redrawn at the position clicked upon by the user and the ratio between the x-coordinate of the cursor position and the width of the valuator is computed. This ratio is treated as the valuator reading. This real number is then passed back to the calling T primitive which subsequently returns the response (Scalar \texttt{<valuator-reading>}) to the I/O system. Since this is a 'success' response, the I/O system invokes the success continuation of the \texttt{readValuator} call with \texttt{<valuator-reading>} as its argument.

4.3.5 Keyboard Related Operations

Keyboard is the physical device that implements the string input logical device functionality. As discussed earlier, the Haskell I/O system uses the function

\[
\texttt{readStrDevice} :: \texttt{Channel} \to \texttt{FailCont} \to \texttt{StrCont} \to \texttt{Dialogue}
\]

to read input from the logical string input device. This function issues the request \texttt{ReadStringDevice} on the continuation stream. The Haskell interpreter, in response to this request on the stream, invokes the following T primitives in sequence:

Step 1: Call \texttt{t\_strDevExists} to check if the channel is valid

This primitive returns \texttt{True} if the channel is valid. Otherwise, it returns \texttt{False}. This function in turn calls a C function, namely \texttt{StrInputExists} with the integer argument \texttt{channel}. \texttt{StrInputExists} checks the array of workstation objects maintained by it
(refer Section 4.3.1) to see if the argument channel points to a valid workstation object. If so, it returns True and so does t_strDevExists. If channel is not an index to a valid workstation object in the array of workstations, StrDevExists returns False, so that t_strDevExists also returns False. If the call to t_strDevExists yields False, the failure continuation of readStrDevice is invoked. Otherwise, the next step is performed.

Step 2: If the channel was found to be valid in Step 1, then call
t_readStrDevice to read a string from the channel

The T primitive t_readStringDevice is invoked with the integer value channel as the argument. The T primitive, in turn, invokes the corresponding C function ReadStrInputDevice to read a string from the keyboard. The function ReadStrInputDevice is called with two arguments: the integer channel argument and a string buffer a sufficient length to hold a long character string. ReadStrInputDevice issues the required Xlib calls to scan the keyboard, wait for keyboard input and copies all the input characters, upto the first carriage return character, into the string buffer. The buffer is then returned to the calling T primitive. t_readStringDevice finally returns to the Haskell system the response containing the string read in.

Step 3: Invoke the success continuation
Haskell I/O system invokes the success continuation of readStrDevice with the string which was just read in as the argument.

4.3.6 Implementation of the Frame Display Operation

This is probably the most important of the graphics I/O functions discussed so far. The Haskell function to display a frame object is:

```
showFrame :: Channel -> Frame -> Window2 -> Window2 -> GraphicsContext ->
          FailCont -> SuccCont -> Dialogue
```

The arguments are the workstation identifier (channel), the frame to be displayed, the world window, the viewport, and the set of attribute-value pairs (type GraphicsContext) pertaining to graphics operations. The last argument contains value settings of such attributes
as the graphics paint mode, whether the screen is to be cleared before drawing, the foreground and background colors, etc..

As mentioned earlier, every workstation has, among other devices, an opened X display connection and an opened X window whose dimensions are the same as the ones specified for the workstation object when it was opened. This window is the display surface and used to output frame objects as Xlib drawings. When a frame is sought to be displayed, the Haskell I/O system breaks the operation into a sequence of Vector2 display operations each of which is ultimately translated into C function calls to perform the respective operations (DrawPoint, DrawLine, DrawArc, DrawLabel).

showFrame issues the request

(DumpFrame args)

to the Haskell I/O system, where args represents the list of arguments passed to showFrame. Upon receipt of this request, the I/O system performs the following operations:

Step 1: Set the graphics attribute values
The Haskell function

setGraphicsContext :: Channel -> Frame -> GraphicsContext -> Window2 -> Boolean

treats the third argument as a list of attribute-value pairs and invokes, for each pair, the corresponding T module to set the value onto the workstation. The frame object to be displayed is also passed to this function since, if it is a null entity (EmptyFrame) the function returns trivially. The last argument is the viewport using which the frame object is to be displayed. When the ClearBeforeDump attribute of the workstation ('clear the display surface before performing a showFrame operation') has been set the viewport and the window are used to compute the section of the display surface that should be cleared.

Should channel not identify a legally initialized workstation object, setGraphicsContext returns False which in turn causes the I/O system to send a failure response to the calling application.
Step 2: Display the frame object

The function

\[
\text{displayFrame} :: \text{Channel} \rightarrow \text{Frame} \rightarrow \text{Window2} \rightarrow \text{Window2} \rightarrow \text{Boolean}
\]

is invoked to display the frame. It takes the workstation identifier \textit{channel}, the frame, the window and the viewport. If the frame is successfully displayed it returns \textit{True}, else it returns \textit{False}. The function operates as follows:

* If the frame argument is the null entity, \textit{EmptyFrame}, it trivially returns \textit{True}.
* If the frame argument is of the form
  \[
  (\text{UnionFrame} \ \text{frame}_1 \ \text{frame}_2)
  \]
  it returns the conjunction of the results of invoking \textit{displayFrame} with \textit{frame}_1 and \textit{frame}_2.
* If the frame argument is of the form
  \[
  (\text{MkFrame} \ \text{vector} \ \text{frame})
  \]
  then
  1. display the Vector2 object, namely, \textit{vector} on the specified workstation (\textit{channel}) using the specified viewport. Ideally, at this point \textit{vector} should be clipped against the specified window. However, in this version the function \textit{clipVector2} has not been implemented and is merely an identity function.
  2. If \textit{vector} was successfully displayed, invoke \textit{displayFrame} recursively to display the rest of the frame, namely, \textit{frame}. Otherwise return \textit{False}.

The Vector2 display function, \textit{displayVector2}, calls the appropriate T primitive (\textit{t_drawPoint}, \textit{t_drawLine}, \textit{t_drawArc} or \textit{t_drawLabel}) for the type of the Vector2 object to be displayed (Point2, Line2, Arc2 or Label). This results in a call to the appropriate C primitive (\textit{DrawPoint}, \textit{DrawLine}, \textit{DrawArc} or \textit{DrawLabel}). The C primitive computes the window coordinates of the specified Vector2 object using the window to viewport transformation commonly used in graphics. Once the actual display coordinates have been computed, the appropriate Xlib calls are issued to draw the figure.
4.4 A Note on the Implementation of Label Type

The root cause of what might appear to be a non-intuitive definition of the Label type lies in the way in which fonts have been implemented in X window system, which provides the system its basic graphics capabilities. The fonts in X (release 11, version 4) are implemented using raster masks of the each character comprising the character set. Hence, the string of a Label object is displayed by displaying in sequence the raster masks of the characters of the string. Thus while it is true of objects of other types (Point2, Line2, etc) that there exists a one-to-one correspondence between the points that define the object on the device coordinate space and the points that make up the object in world coordinate space, this cannot be said of the Label type objects. This is because the points that make up the object in world coordinate space are not defined.

This affects the pickFrame operation in a crucial way. As discussed in Chapter III, pickFrame reads the locator device and converts the point read in from normalized device coordinates to the world coordinates using the window and viewport values used for display. Having converted the point to the world coordinate space, the function then checks to see if the point (type Point2) lies in any of the objects (type Point2, Line2, Arc2 or Label) forming the frame under consideration. Since the Point2, Line2 and Arc2 type objects have clear geometric definitions, it is possible to define an algorithm to check if a given point in world coordinate space lies on the object. However, no such algorithm can be designed for the Label data type. Hence the point returned by the locator must be checked to see if it lies on the label raster in device coordinate space. This requires that the origin of the Label raster (i.e., the position of the Label object) be transformed to the device coordinate space. Thus, the membership check function needs to know the window and viewport using which the Label object was displayed. Consequently, every Label object has been defined to contain the window and the viewport using which it is to be displayed.
Chapter V

Conclusions and Future Work

5.1 Summary of Results

This project was commenced with the belief that the declarative programming style might be suitable for graphics computation. We sought to provide a part of the functionality provided by common graphics functional standards such as GKS or PHIGS, including picture segmentation, a uniform set of drawing primitives, interface to logical devices and workstation management functions. Another important goal of the project was to use the framework as a testbed for continuation I/O.

Using the purely functional language Haskell, we have defined and implemented the basic set of data types and primitives required for graphics. We have also developed a structured hierarchy of types and classes. This hierarchy includes the most primitive types required for general purpose computation as well as the complex types required for graphics programming. While we developed the type structure for both 2D and 3D graphics, our implementation does not include the structures defined for 3D graphics. To establish the graphics I/O framework, we have defined the functions of the six logical graphics devices and extended the Haskell I/O system by defining the application interface to these devices using continuation I/O model. In order to provide the system its basic graphics capabilities, we have interfaced the Haskell system with the X Window System using the T foreign function interface. The logical device functionality was realized in this layer of the system (the X layer).

We have written graphics applications using the framework, varying from the very simple to more complex ones, to gauge the usability and the execution efficiency of the system. The most significant observations are discussed in the following section.

5.2 Concluding Remarks

We conclude that using functional programming for graphics software construction presents advantages as well as disadvantages. The advantages include the improved structure of programs and a greater potential for development of program verification mechanisms. The disadvantages include the effect of lazy evaluation on graphical user interfaces and the reduced performance of
applications during garbage collection. Further, there are advantages to using our framework for graphics program development instead of using a purely functional programming language: our framework provides to the user not only all the features of a pure functional programming environment, but also the types and I/O mechanisms that are necessary to write applications for graphics.

On the whole, we conclude that if the language interpreter and the garbage collection algorithms are implemented efficiently, functional graphics systems can be made practical. Specific observations are summarised in the following sections.

5.2.1 Readability and Structure of Programs

While arguments emphasizing program readability or lack thereof are subjective by nature, a purely functional programming system can lay a more concrete claim to readability mainly for three reasons: (1) the static nature of the program and (2) the absence of global variable and (3) the absence of side effects.

Conventional programmers find programs written in a purely functional style difficult to read because the statements are not necessarily laid out in the order in which they are executed. Continuation I/O model allows us to write applications so that the I/O statements can be laid out in the sequence in which the flow of events takes place during execution. This makes it less confusing for a programmer who is used to imperative style of programming to read and understand programs written in FunGraf. However, even using FunGraf we found that program readability decreased as program complexity increased.

5.2.2 Problems caused by Lazy Evaluation in Graphical User Interfaces

Lazy evaluation sometimes hampers the synchronization between the order in which the function execution requests are made by the application and the order in which they are executed on the workstation. This is rooted in the problem that while functional programming does not recognize side effects, the essence of input-output operations is side effects. For instance, in performing a sequence of graphics I/O operations, the programmer expects to provide a visual effect using the first I/O operation and modify this effect using the subsequent operations. The reader may note that this can be done even if the programmer preserves referential transparency in the domain of the programming language. The essence of the problem is that while the application may not need the data returned by the execution of the function, it may
need the effect of the function execution on the display surface.

The problem can be better illustrated by two examples that we encountered in experimenting with the system.

Example 1: Need for an Explicit Evaluation followed by openWorkStation Request

On most graphics platforms, the operation of opening and initializing the workstation is accompanied by a visual feedback on the display surface. It may be merely the clearing of the display surface or the appearance of special windows and new colors on the surface. The openWorkStation function call in the FunGraf system causes an X display connection to be set up, the display window to appear and if selected, the valuator to appear on the display surface. Consider the program fragment in Figure 5.1 [a]. The program does the following:

a) opens workstation A
b) opens workstation B
c) performs an input operation on workstation B
d) displays a frame \( f \) on workstation A

Due to lazy evaluation, the order of events seen on the workstations is different from the order which the programmer apparently intended in writing this application. The application requests the Haskell evaluator or the interpreter to open and initialize the workstation A. The result of this call is a channel identifier, say channelA. Since the application does not immediately demand the value of channelA, the function is not executed by the interpreter. Subsequently the application requests that workstation B be opened and initialized and commences an input operation on the workstation (using the channel id channelB returned by the second openWorkStation request). Since the value of channelB is immediately required, the openWorkStation request on B is immediately executed and the input performed. The input operation performed by the application on channelB cannot be completed without the user operating the input device. Finally, the program gets around to displaying the frame \( f \) on the workstation A. Since the value of channelA is now demanded by the application, the openWorkStation request on workstation A that was pending is now executed and the frame displayed. Hence, the user of the application does not see the frame \( f \) on A until he or she inputs the appropriate data in workstation B to complete the input operation on that workstation, a sequence of operations completely different from that intended by the author of the program.
[a] Original Program Fragment

openWorkStation A attributes ; returns channel id channelA

errorHandler ; Handle errors with opening workstation A

\( \text{\textbackslash channelA} \rightarrow \text{openWorkStation B attributes} \)

; returns channel id channelB

errorHandler

\( \text{\textbackslash channelB} \rightarrow \)

{ ... input operations on channel channelB ... } ; Step 1

{ ... display Frame f on channel channelA ... } ; Step 2

)

[b] Modified Program

openWorkStation A attributes ; returns channel id channelA

errorHandler ; Handle errors with opening workstation A

\( \text{\textbackslash channelA} \rightarrow \)

if (channelA > 0) ; explicit evaluation to force operation

( openWorkStation B attributes

; returns channel id channelB

errorHandler

\( \text{\textbackslash channelB} \rightarrow \)

{ ... input operations on channel channelB ... } ; Step 1

{ ... display Frame f on channel channelA ... } ; Step 2

)

else (... )

Figure 5.1 : Need for an Explicit Evaluation to Sidestep Lazy Evaluation
In order to get around this problem, the program must explicitly demand the value of the \textit{openWorkStation} operation on A before proceeding with other operations (Figure 5.1 [b]).

**Example 2: Miscoordination between input and output operations caused by lazy evaluation**

Friendly user interface programs often interleave their input commands (requiring input from the user) with output commands that display instructions as to how the input data is to be supplied. While input commands cannot be completed until the user performs the input operation, output commands that do not depend on input data can be completed without any action on user's part. Consequently, when input and output commands are interleaved, the input operations are subject to delayed evaluation while the output operations are executed immediately.

This manifests as a practical problem in graphical user interfaces which are inherently repetitive, resulting in the function implementing the GUI invoking itself recursively. Consider for example the following situation.

It is required of a friendly user interface to inform the user that an input is expected before commencing the operation. In following this maxim, an application that we wrote using FunGraf uses a "notice board" mechanism as follows:

1. The application opens the intended workstation with the user-desired workstation dimensions (channel 1).
2. The application opens another connection to the same physical workstation to provide a notice board function (channel 2).

The display window associated with channel 1 is used for performing the regular graphics display operations and that associated with channel 2 is used to display character strings as instructions to the user. These instructions describe succinctly the operations that are currently in progress and the type of response expected from the user. In one cycle of the user interface, the notice displays the instructions for the user to input a certain value in channel 1 and reads input from channel 1. Consider two successive cycles:

1. Display message X on channel 2
2. Wait for user input on channel 1
3. Display message Y on channel 2
4. Wait for input on channel 1
The effect of this is that message X is displayed on the notice board and the input operation is commenced. Due to lazy evaluation, the execution does not wait for the input operation to be over. Thus the message Y is overwritten on the message X on the notice board during the input operation. This results in a confusing user interface. Thus in order to create realistic user interfaces, lazy evaluation must be suppressed to synchronize with user actions.

5.2.3 Effect of Garbage Collection and Software Layering on the Performance

As discussed in Chapter 4 graphics operations consume a lot of heap space which must eventually be reclaimed by the garbage collector. When the garbage collector is invoked, the execution of an application seems to "freeze" midway and the events generated by the user with the input devices seem to have no effect on the display output (Figure 5.2). This operation normally takes a few seconds (as observed on the Yale Haskell system) after which the application "comes alive" and the events that the user had generated during the garbage collection process appear to be processed all at once. This is similar to the freeze of workstation operations in a vector scan graphics system when the workstation begins garbage collection and compaction of the memory blocks allocated to deleted display lists [Foley and van Dam 1988]. Since the application receives no feedback from the system as to the commencement of the garbage collection, it is not possible for the program to warn the user of the application that garbage collection is about to begin. This leads to inelegant breaks in the user interface.

The effect of software layering becomes evident when the application is dealing with complex frames. Since the display of a complex picture is broken down into a serial display of its simple component vectors, operations that call for rapid rotation/translation/-scaling of complex pictures exhibit a notable delay in repainting the picture, and the process of repainting the component vectors becomes easily observable. This was clearly observed in running the module Menu [program "menu.has"] (Appendix B) with a the image of a landscape (a frame containing dozens of Vector2 objects).

Unless we get around these two problems, the possible use of FunGraf for any practical real-time graphics application would remain in question. While vector scan systems implement frame display operation similarly (that is, by breaking up the display of a frame into a sequence of primitive display operations), the primitive display operation is implemented in hardware. This considerably enhances the speed with which complex pictures can be drawn. Similarly, to improve the performance of functional graphics systems, the software layer that displays
Rubberbanding Demo

 Remarks

The program monitors the current position and draws a "rubberbanding" line from a user preselected anchor point and the current cursor point. When the Haskell system initiates the garbage collection process, the display seems to freeze and the program seems not to respond to user generated events as the cursor moves ahead, leaving behind the line. Refer to Appendix B.

Figure 5.2 Freezing of Display during Garbage Collection
primitives can be replaced with a functionally equivalent hardware system.

Overhead incurred due to runtime garbage collection can seriously affect the usability of purely functional languages for the implementation of practical real-time systems. With the use of highly efficient garbage collection algorithms, functional graphics systems can be made practical.

5.3 Future Work

This project was commenced with a broad set of objectives. These included exploring the use of functional programming for computer graphics software construction, experimenting with the new models for functional input-output and exploring the avenues available for program correctness verification. While we have achieved a modest amount of success, the present research can be continued in the following directions.

5.3.1 Graphics Input-Output using Stream Model and Systems

Model

We chose to use the continuation model for implementing functional graphics input-output. While the results were encouraging in terms of programmer acceptability and versatility to handle the different types of input and output required by required by graphics, the same experiments must be conducted with stream I/O model and systems model [Hudak and Sundaresh 1988]. Although Hudak and Sundaresh have established the equivalence of the three models, one model may have stylistic advantages for certain applications.

Streams are lazy lists of I/O requests (sent by the application) and responses (returned by the operating system). The application sends requests for performing I/O operations to the operating system which in turn returns to the application appropriate responses indicating whether or not the operations were completed successfully. This form of communication between two entities wherein one entity (client) requests a service and the other (server) responds is used extensively in windowing systems and data communications. For instance, the protocol of communication between an X application and the X server is quite similar to that between a Haskell application and the operating system in the stream I/O model. Hence it would be interesting to use stream model for functional graphics systems built on windowing systems.

System model is interesting because of its simplicity and hence must be experimented with in future versions of FunGraf. Apart from simplicity, another advantage of system model over the stream model is that the tedium of matching requests and responses is avoided. However,
modeling the effect of nondeterministic interleaving of execution of different programs on the
operating system imposes expensive overheads on the language implementation [Hudak and
Sundaresh 1988].

5.3.2 Use of Boolean Operations and Application to 3D graphics

In solid modelling application, one would like to model real life objects using program
abstractions and perform operations such as gluing, "chopping off" and finding the common
components. Currently, the graphics functional standards do not provide all the boolean
operations and a layer of software has to be written on top of the GFS to provide the end users
this feature. We have suggested a uniform way to define such operations (union, intersection and
subtraction) within the graphics functional standard for the primitive objects which can be
systematically extended to top level abstractions (frames and scenes).

Further, most practical graphics applications deal with a three dimensional world
coordinate system. The implementation of the given system can be, with some effort, extended to
the three dimensional domain.
REFERENCES


Appendix A

Formal Description of the Framework

The purpose of this appendix is to formally present the extensions made to Yale Haskell to arrive at the FunGraf framework. The reader is expected to be familiar with the syntax of Yale Haskell. The description is given in three parts.

Part I describes the types and basic structures forming the framework.

Part II briefly lists the workstation-related attributes used in any graphics programming system.

Part III shows formally the different I/O request functions, written in continuation style, that form the complete I/O system from the programmer's perspective.

Part I

The formal definitions of the types and operations on the types that form the basis of the framework are shown in the attached program listing. These definitions are contained in the module PreludeMy (file PreludeMy.hs) and are included in the Haskell system when the module is compiled into the Haskell Prelude. The definitions start with the most primitive type, namely, the two dimensional point (Pt2).

Part II

This part shows formally the various attribute values of a workstation that may be used in writing a graphics program using the framework. These attributes include the cursor types, whether valuator is required, name of the workstation, etc. These are definitions are used in making graphics I/O requests.

Part III

This formally presents the I/O functions that form the Yale Haskell I/O system. For the sake of completeness, the original system (i.e., the functions for the normal I/O operations) have also been included in the description.
module PreludeMy where

-- INCLUDE DEFINITIONS FROM THE FOLLOWING PRE-DEFINED MODULES:
import PreludeRealCore
import PreludeTextCore
import PreludeNumeric
import PreludeFloating
import PreludeFloat
import PreludeFloatIO
import PreludeString
import PreludeExtIO

infix @@

-- DEFINITION OF TWO DIMENSIONAL POINT:
data Point2 = EmptyPoint2 | (Pt2 Float Float) deriving ()

-- OPERATIONS ON THE TWO DIMENSIONAL POINT:

-- SCALING A 2D POINT
scalePt2 :: Point2 -> Float -> Point2
scalePt2 EmptyPoint2 _ = EmptyPoint2
scalePt2 _ 0.0 = EmptyPoint2
scalePt2 (Pt2 x y) s = (Pt2 (s*x) (s*y))

-- TRANSLATING A 2D POINT
movePt2 :: Point2 -> Point2 -> Point2
movePt2 EmptyPoint2 _ = EmptyPoint2
movePt2 (Pt2 x y) EmptyPoint2 = (Pt2 x y)
movePt2 (Pt2 x y) (Pt2 tox toy) =
       (Pt2 xx yy)
where
          xx = if (tox >= 0.0) then
               (x + tox)
             else
               (x - (my_abs tox))
          yy = if (toy >= 0.0) then
               (y + toy)
             else
               (y - (my_abs toy))

-- ROTATING A 2D POINT ABOUT ORIGIN
rotatePt2Origin :: Point2 -> Float -> Point2
rotatePt2Origin EmptyPoint2 _ = EmptyPoint2
rotatePt2Origin (Pt2 x y) 0.0 = (Pt2 x y)
rotatePt2Origin (Pt2 x y) angle =
       (Pt2
          (x*(cos angle) - y*(sin angle))
          (x*(sin angle) + y*(cos angle))
       )

-- ROTATING A 2D POINT ABOUT AN ARBITRARY POINT
rotatePt2 :: Point2 -> Point2 -> Float -> Point2
rotatePt2 (Pt2 x y) _ 0.0 = (Pt2 x y)
rotatePt2 (Pt2 x y) EmptyPoint2 ___ = (Pt2 x y)
rotatePt2 EmptyPoint2 ___ = EmptyPoint2
rotatePt2 (Pt2 x y) (Pt2 px py) angle = movePt2
  {rotatePt2Origin
   (Pt2 (x - px) (y - py))
   angle
  }
  (Pt2 px py)

-- DEFINITION OF 2 DIMENSIONAL LINE
data Line2 = EmptyLine2 | (MkLine2 Point2 Point2) deriving ()

-- OPERATIONS ON THE 2 DIMENSIONAL LINE

-- TRANSLATING A 2D LINE
moveLine2 :: Line2 -> Point2 -> Line2
moveLine2 EmptyLine2 ___ = EmptyLine2
moveLine2 (MkLine2 start end) EmptyPoint2 = (MkLine2 start end)
moveLine2 (MkLine2 (Pt2 sx sy) (Pt2 ex ey)) (Pt2 tox toy) =
  (MkLine2
   (movePt2 (Pt2 sx sy) (Pt2 tox toy))
   (movePt2 (Pt2 ex ey) (Pt2 tox toy)))

-- SCALING A 2D LINE
scaleLine2 :: Line2 -> Float -> Line2
scaleLine2 EmptyLine2 ___ = EmptyLine2
scaleLine2 0.0 = EmptyLine2
scaleLine2 (MkLine2 start end) factor =
  (MkLine2 (scalePt2 start factor)
   (scalePt2 end factor))

-- ROTATING A 2D LINE ABOUT A GIVEN POINT BY A GIVEN ANGLE
rotateLine2 :: Line2 -> Point2 -> Float -> Line2
rotateLine2 EmptyLine2 ___ = EmptyLine2
rotateLine2 (MkLine2 start end) _ 0.0 = (MkLine2 start end)
rotateLine2 (MkLine2 start end) EmptyPoint2 ___ = (MkLine2 start end)
rotateLine2 (MkLine2 start end) pivot angle =
  (MkLine2 (rotatePt2 start pivot angle)
   (rotatePt2 end pivot angle))

-- DEFINITION OF 2 DIMENSIONAL ARC
data Arc2 = EmptyArc2 | (MkArc2 Point2 Float Float Float) deriving ()
-- Arc2 is identified by the centre, the Radius, the angle of one end
-- of the arc relative to the 3'O clock position and the
-- counterclockwise sweep of second angle (negative == clockwise)
-- from the first point. All angles are in degrees.

-- OPERATIONS ON THE 2 DIMENSIONAL ARC

-- TRANSLATING THE GIVEN 2D ARC
moveArc2 :: Arc2 -> Point2 -> Arc2
moveArc2 EmptyArc2 _ = EmptyArc2
moveArc2 (MkArc2 centre radius angle1 sweep) EmptyPoint2 =
    (MkArc2 centre radius angle1 sweep)
moveArc2 (MkArc2 centre radius angle1 sweep) (Pt2 x y) =
    (MkArc2 (movePt2 centre (Pt2 x y)) radius angle1 sweep)

-- SCALING THE GIVEN 2D ARC
scaleArc2 :: Arc2 -> Float -> Arc2
scaleArc2 EmptyArc2 _ = EmptyArc2
scaleArc2 _ 0.0 = EmptyArc2
scaleArc2 (MkArc2 centre radius angle1 sweep) s =
    (MkArc2 (scalePt2 centre s) (radius*s) angle1 sweep)

-- ROTATING A 2D LINE ABOUT A GIVEN POINT BY A GIVEN ANGLE
rotateArc2 :: Arc2 -> Point2 -> Float -> Arc2
rotateArc2 EmptyArc2 _ _ = EmptyArc2
rotateArc2 (MkArc2 centre radius angle1 sweep) _ _ 0.0 =
    (MkArc2 centre radius angle1 sweep)
rotateArc2 (MkArc2 centre radius angle1 sweep) EmptyPoint2 _ _ =
    (MkArc2 centre radius angle1 sweep)

-- (THE ANGLE OF THE ARC IS NOT AFFECTED BY ROTATION).
rotateArc2 (MkArc2 centre radius angle1 sweep) pivot angle _ =
    (MkArc2 (rotatePt2 centre pivot angle) radius angle1 sweep)

-- DEFINITION OF LABEL PRIMITIVE:
data Label = EmptyLabel | (MkLabel String Point2 Window2 Window2)
    deriving ()

-- OPERATIONS ON THE LABEL PRIMITIVE:
moveLabel :: Label -> Point2 -> Label
moveLabel EmptyLabel _ = EmptyLabel
moveLabel x EmptyPoint2 = x
moveLabel (MkLabel txt_string point w v) toa =
    MkLabel txt_string (movePt2 point toa) w v

-- (Labels cannot be scaled in our system (raste Fonts), except in
-- trivial case)
scaleLabel :: Label -> Float -> Label
scaleLabel _ 0.0 = EmptyLabel
scaleLabel x _ = x

-- The text string CANNOT be rotated about any pivot.
rotateLabel :: Label -> Point2 -> Float -> Label
rotateLabel x _ _ = x

-- GENERALIZED 2 DIMENSIONAL VECTOR DESCRIPTION ::

-- A GEN VECTOR CAN BE ONE OF : A POINT, A LINE, AN ARC or A LABEL
data Vector2 = EmptyVector2
    | (Pt2ToVector2 Point2)
    | (Line2ToVector2 Line2)
    | (Arc2ToVector2 Arc2)
    | (LabelToVector2 Label) deriving ()
-- OPERATIONS ON THE GENERALIZED 2 DIMENSIONAL VECTOR ARE DEFINED
-- IN TERMS OF THE CORRESPONDING OPERATION ON THE POINT/LINE/ARC/
-- /LABEL FORMING THE VECTOR.

moveVector2 :: Vector2 -> Point2 -> Vector2
moveVector2 EmptyVector2 _ = EmptyVector2
moveVector2 (Pt2ToVector2 point) toa =
    Pt2ToVector2 (movePt2 point toa)
moveVector2 (Line2ToVector2 line) toa =
    Line2ToVector2 (moveLine2 line toa)
moveVector2 (Arc2ToVector2 arc) toa =
    Arc2ToVector2 (moveArc2 arc toa)
moveVector2 (LabelToVector2 label) toa =
    LabelToVector2 (moveLabel label toa)

scaleVector2 :: Vector2 -> Float -> Vector2
scaleVector2 EmptyVector2 _ = EmptyVector2
scaleVector2 (Pt2ToVector2 point) s =
    Pt2ToVector2 (scalePt2 point s)
scaleVector2 (Line2ToVector2 line) s =
    Line2ToVector2 (scaleLine2 line s)
scaleVector2 (Arc2ToVector2 arc) s =
    Arc2ToVector2 (scaleArc2 arc s)
scaleVector2 (LabelToVector2 label) s =
    LabelToVector2 (scaleLabel label s)

rotateVector2 :: Vector2 -> Point2 -> Float -> Vector2
rotateVector2 EmptyVector2 _ = EmptyVector2
rotateVector2 (Pt2ToVector2 point) pivot angle =
    Pt2ToVector2 (rotatePt2 point pivot angle)
rotateVector2 (Line2ToVector2 line) pivot angle =
    Line2ToVector2 (rotateLine2 line pivot angle)
rotateVector2 (Arc2ToVector2 arc) pivot angle =
    Arc2ToVector2 (rotateArc2 arc pivot angle)
rotateVector2 (LabelToVector2 label) pivot angle =
    LabelToVector2 (rotateLabel label pivot angle)

-- DEFINITION OF A 2D PICTURE MODULE ("Frame"):

-- FRAME IS THE 2D PICTURE MODULE: COLLECTION OF GENERALIZED VECTORS.
data Frame = EmptyFrame
    | (MkFrame Vector2 Frame)
    | (UnionFrame Frame Frame)
    | (IntersectFrame Frame Frame)
    | (FrameDifference Frame Frame) deriving ()

-- OPERATIONS ON DEFINED ON FRAME ARE DEFINED IN TERMS OF THOSE ON
-- THE GENERALIZED VECTORS COMPRISING THE FRAME.
moveFrame :: Frame -> point2 -> Frame
moveFrame EmptyFrame _ = EmptyFrame
moveFrame frame EmptyPoint2 = frame
moveFrame (MkFrame v f) toa = MkFrame (moveVector2 v f toa)
moveFrame (UnionFrame f1 f2) toa =
    UnionFrame (moveFrame f1 toa)

91
moveFrame (IntersectFrame f1 f2) toa =
    IntersectFrame (moveFrame f1 toa)
    (moveFrame f2 toa)

moveFrame (FrameDifference f1 f2) toa =
    FrameDifference (moveFrame f1 toa)
    (moveFrame f2 toa)

-- Scale a Frame about origin ::

scaleFrame :: Frame -> Float -> Frame
scaleFrame EmptyFrame _ = EmptyFrame
scaleFrame frame 0.0 = frame
scaleFrame (MkFrame v f) s = MkFrame (scaleVector2 v s)
    (scaleFrame f s)

scaleFrame (UnionFrame f1 f2) s =
    UnionFrame (scaleFrame f1 s) (scaleFrame f2 s)

scaleFrame (IntersectFrame f1 f2) s =
    IntersectFrame (scaleFrame f1 s) (scaleFrame f2 s)

scaleFrame (FrameDifference f1 f2) s =
    FrameDifference (scaleFrame f1 s) (scaleFrame f2 s)

-- Rotate a Frame about a given pivot (point) ::

rotateFrame :: Frame -> Point2 -> Float -> Frame
rotateFrame EmptyFrame _ _ = EmptyFrame
rotateFrame frame _ 0.0 = frame
rotateFrame frame EmptyPoint2 _ = frame
rotateFrame (MkFrame v f) pivot angle =
    MkFrame (rotateVector2 v pivot angle)
    (rotateFrame f pivot angle)
rotateFrame (UnionFrame f1 f2) pivot angle =
    UnionFrame (rotateFrame f1 pivot angle)
    (rotateFrame f2 pivot angle)
rotateFrame (IntersectFrame f1 f2) pivot angle =
    IntersectFrame (rotateFrame f1 pivot angle)
    (rotateFrame f2 pivot angle)
rotateFrame (FrameDifference f1 f2) pivot angle =
    FrameDifference (rotateFrame f1 pivot angle)
    (rotateFrame f2 pivot angle)

-- USE OF COMPOSITE DATA TYPE (Window/Viewport definition):
data Window2 = (MkWindow2 Point2 Point2) deriving ()

-----

-- DEFINITION OF POINT CLASS:
class Point a where
    (@@) :: a -> a -> a

instance Point Point2 where
    x @@ y = addPt2 x y

addPt2 :: Point2 -> Point2 -> Point2
addPt2 (Pt2 a b) (Pt2 c d) = Pt2 (a+c) (b+d)
-- ATTRIBUTES SPECIFIED DURING DISPLAY OPERATIONS

data Modes = MCopy
          | MXor
          | MClear
          | (Fill display with zeros)
          deriving ()

data AttributeValue = (ClearBeforeDump Bool)
                      | (WriteMode Modes)
                      | (Use mode "mode" while drawing)
                      deriving ()

type GraphicsContext = [AttributeValue]

-- BUNDLED ATTRIBUTES USED WHILE OPENING A WORKSTATION ::

data BundleElements = (ValuatorNeeded Int)
                      | (WsWidth Int)
                      | (WsHeight Int)
                      | (WsName String)
                      | (Specify workstation name)
                      deriving ()

type BundledAttributes = [BundleElements]

-- DIFFERENT TYPES OF CURSORS AVAILABLE ON WORKSTATIONS

data Cursor = CUR_DANGER
             | CUR_PICK
             | CUR_CLICK
             | CUR_PLACE
             | CUR_CROSS
             | CUR_PENCIL
             | CUR_CIRCLE
             | CUR_DEFAULT deriving ()
module PreludeIOCore where

import PreludeRealCore

-- IMPORT DEFINITIONS CREATED FOR GRAPHICS EXTENSION FROM THE GRAPHICS
-- TYPE EXTENSION PRELUDE
import PreludeMy(Point2, Frame, Window2, GraphicsContext, BundleAttributes, Cursor)

-- FILE AND CHANNEL NAMES:

 type Name ~ String
 stdin   = "stdin"
 stdout  = "stdout"
 stderr  = "stderr"
 stdecho = "stdecho"

-- VARIOUS TYPES OF REQUESTS AND RESPONSES:

 data Request = -- file system requests:
 | ReadFile    Name
 | WriteFile   Name String
 | AppendFile  Name String
 | ReadBinFile Name
 | WriteBinFile Name Bin
 | AppendBinFile Name Bin
 | DeleteFile  Name
 | StatusFile  Name

-- channel system requests:
 | ReadChan    Name
 | AppendChan  Name String
 | ReadBinChan Name
 | AppendBinChan Name Bin
 | StatusChan  Name

-- GRAPHICS REQUESTS (EXTENSION):

-- graphics channel system requests:
 | OpenWS String BundleAttributes
 | CloseWS Channel
 | DumpFrame Channel Frame Window2 Window2 GraphicsContext
 | ReadLocator Channel Cursor
 | ReadStroke Channel
 | PickFrame Channel Cursor Window2 Window2 [Frame]
 | ReadValuator Channel
 | ReadStringDevice Channel

-- environment requests:
 | Echo Bool
 | GetArgs
 | GetEnv Name
 | SetEnv Name String deriving ()
data Response = Success
  | Str String
  | Bin Bin

-- GRAPHICS RESPONSES (EXTENSION):
  | Ws Channel
  | LocFr Point2
  | Scalar Float
  | Fr Frame
  | Failure IOError deriving ()

data IOError = WriteError String
  | ReadError String
  | SearchError String
  | FormatError String

--- GRAPHICS I/O ERRORS ARE FORMATTED AS "OtherError" TYPE:
  | OtherError String deriving ()

-- CONTINUATION-BASED I/O:

type Dialogue = [Response] -> [Request]
type SuccCont = Dialogue
type StrCont = String -> Dialogue
type BinCont = Bin -> Dialogue
type FailCont = IOError -> Dialogue

-- GRAPHICS I/O RESPONSES TAGS (EXTENSION):
type OpenCont = Channel -> Dialogue
type LocatorCont = Point2 -> Dialogue
type ValuatorCont = Float -> Dialogue
type FrameCont = Frame -> Dialogue

-- THIS PRELUDE MODULE DEScribes THE COMPLETE HASKELL I/O FRAMEWORK
-- INCLUDES THE EXTENSIONS MADE FOR GRAPHICS I/O
module PreludeContIO where

import PreludeRealCore
import PreludeIOCore
import PreludeTextCore
-- (IMPORT DEFINITIONS CREATED FOR GRAPHICS EXTENSION FROM THE
-- GRAPHICS TYPE EXTENSION PRELUDE)
import PreludeMy

done :: Dialogue
readFile :: Name -> FailCont -> StrCont -> Dialogue
writeFile :: Name -> String -> FailCont -> SuccCont -> Dialogue
appendFile :: Name -> String -> FailCont -> SuccCont -> Dialogue
readBinFile :: Name -> FailCont -> BinCont -> Dialogue
writeBinFile :: Name -> Bin -> FailCont -> SuccCont -> Dialogue
appendBinFile :: Name -> Bin -> FailCont -> SuccCont -> Dialogue
deleteFile :: Name -> FailCont -> SuccCont -> Dialogue
statusFile :: Name -> FailCont -> StrCont -> Dialogue
readChan :: Name -> FailCont -> StrCont -> Dialogue
appendChan :: Name -> String -> FailCont -> SuccCont -> Dialogue
readBinChan :: Name -> FailCont -> BinCont -> Dialogue
appendBinChan :: Name -> Bin -> FailCont -> SuccCont -> Dialogue
echo :: Bool -> FailCont -> SuccCont -> Dialogue
getArgs :: FailCont -> StrCont -> Dialogue
getEnv :: Name -> FailCont -> StrCont -> Dialogue
setEnv :: Name -> String -> FailCont -> SuccCont -> Dialogue

--- DECLARATIONS AND DEFINITIONS OF GRAPHICS I/O FUNCTIONS
--- (EXTENSION):

--- (...DISPLAY THE SPECIFIED FRAME ON THE SPECIFIED CHANNEL USING
--- THE SPECIFIED ATTRIBUTES)
showFrame :: Channel -> Frame -> Window2 -> Window2 ->
          GraphicsContext -> FailCont -> SuccCont ->
          Dialogue
showFrame channel frame window viewport context fail succ resps =
          (DumpFrame channel frame window viewport context) :
          succDispatch fail succ resps

--- (...OPEN THE NAMED WORKSTATION USING SPECIFIED WORKSTATION
--- ATTRIBUTES)
openWorkStation :: String -> BundledAttributes -> FailCont ->
                    OpenCont -> Dialogue
openWorkStation ws_name ws_attributes fail succ resps =
          (OpenWS ws_name ws_attributes) : wsDispatch fail succ resps

--- (...CLOSE THE SPECIFIED WORKSTATION)
closeWorkStation :: Channel -> FailCont -> SuccCont -> Dialogue
closeWorkStation channel fail succ resps =
          (CloseWS channel) : succDispatch fail succ resps

--- (...READ A POINT FROM THE LOCATOR DEVICE ON THE SPECIFIED CHANNEL)
readLocator :: Channel->Cursor -> FailCont -> LocatorCont ->
              Dialogue
readLocator channel cursor fail succ resps =
          (ReadLocator channel cursor) : locDispatch fail succ resps

--- (...READ A SET OF POINTS FROM THE STROKE DEVICE ON THE SPECIFIED CHANNEL)
readStroke :: Channel -> FailCont -> LocatorCont -> Dialogue
readStroke channel fail succ resps =
          (ReadStroke channel) : locDispatch fail succ resps

--- (...READ A SCALAR FROM THE VALUATOR DEVICE ON THE SPECIFIED CHANNEL)
readValuator :: Channel -> FailCont -> ValuatorCont -> Dialogue
readValuator channel fail succ resps =
          (ReadValuator channel) : valDispatch fail succ resps

--- (...READ A CHARACTER STRING FROM THE STRING INPUT DEVICE ON THE
--- SPECIFIED CHANNEL)
readStringDevice :: Channel -> FailCont -> StrCont -> Dialogue
readStringDevice channel fail succ resps =
(ReadStringDevice channel) : strDispatch fail succ resps

--- (...READ A FRAME FROM THE PICK DEVICE OF THE SPECIFIED CHANNEL):
pickFrame :: Channel -> Cursor -> Window2 -> Window2 -> [Frame] -> FailCont -> FrameCont -> Dialogue
pickFrame channel cursor window viewport frame_list fail
  succ resps =
    (PickFrame channel cursor window viewport frame_list) : frameDispatch fail succ resps
done resps = []

readFile name fail succ resps =
  (ReadFile name) : strDispatch fail succ resps
writeFile name contents fail succ resps =
  (WriteFile name contents) : succDispatch fail succ resps
appendFile name contents fail succ resps =
  (AppendFile name contents) : succDispatch fail succ resps
readBinFile name fail succ resps =
  (ReadBinFile name) : binDispatch fail succ resps
writeBinFile name contents fail succ resps =
  (WriteBinFile name contents) : succDispatch fail succ resps
appendBinFile name contents fail succ resps =
  (AppendBinFile name contents) : succDispatch fail succ resps
deleteFile name fail succ resps =
  (DeleteFile name) : succDispatch fail succ resps
statusFile name fail succ resps =
  (StatusFile name) : strDispatch fail succ resps
readChan name fail succ resps =
  (ReadChan name) : strDispatch fail succ resps
appendChan name contents fail succ resps =
  (AppendChan name contents) : succDispatch fail succ resps
readBinChan name fail succ resps =
  (ReadBinChan name) : binDispatch fail succ resps
appendBinChan name contents fail succ resps =
  (AppendBinChan name contents) : succDispatch fail succ resps
echo bool fail succ resps =
  (Echo bool) : succDispatch fail succ resps
getArgs fail succ resps =
  GetArgs : strDispatch fail succ resps
getEnv name fail succ resps =
  (GetEnv name) : strDispatch fail succ resps
setEnv name val fail succ resps =
    (SetEnv name val) : succDispatch fail succ resps

strDispatch fail succ (resp:resps) = case resp of
    Str val  -> succ val resps
    Failure msg -> fail msg resps

binDispatch fail succ (resp:resps) = case resp of
    Bn val   -> succ val resps
    Failure msg -> fail msg resps

-- MY CODE (S Ramakrishnan)
wsDispatch fail succ (resp:resps) = case resp of
    Ws channel -> succ channel resps
    Failure msg -> fail msg resps

locDispatch fail succ (resp:resps) = case resp of
    LocPr point -> succ point resps
    Failure msg -> fail msg resps

valDispatch fail succ (resp:resps) = case resp of
    Scalar value -> succ value resps
    Failure msg -> fail msg resps

frameDispatch fail succ (resp:resps) = case resp of
    Fr frame    -> succ frame resps
    Failure msg -> fail msg resps

succDispatch fail succ (resp:resps) = case resp of
    Success    -> succ resps
    Failure msg -> fail msg resps

abort :: FailCont
abort msg = done
Appendix B

Illustrative Programs

This appendix presents some programs written using the FunGraf framework illustrating how the framework may be used for implementing some common graphics tasks. The programs have been kept as simple as possible while still illustrating the the features of the system. Every program description is presented with a listing of the program and an X window dump of the output generated by the program. The following programs are included in this appendix:

1. Program to demonstrate interactive picture scaling
2. Program to demonstrate the "rubberbanding" operation.
3. Program to demonstrate the use of the string input device.
4. Program to demonstrate handling of intensive user-interaction ("paintbrush" utility).
5. A simple, menu-based landscape synthesis program.

The descriptions follow in the following sections.
1. Program to Demonstrate Interactive Picture Scaling

   Altering the shape and size of the displayed image based on data interactively supplied by the user is an important feature in many graphics and solid modelling packages. The following simple program demonstrates how this functionality may be implemented using the FunGraf framework described in this report. The main features of FunGraf introduced by this program are as follows:

   * valuater device
   * continuation I/O style for
     - opening, initializing and closing a workstation
     - implementing repetitive user-interface actions (while loops)
   * description of simple pictures using the types of the FunGraf framework (Frames).

   When the program is executed, evaluation begins at function main. Function main opens the workstation identified by the identifier workstation using the workstation attribute VALUATOR_NEEDED (set to True). If an error is encountered in opening the workstation, the error handler errorHandler is invoked; it prints the name of the error condition and aborts the program execution. If the workstation was successfully opened, the channel identification associated with the workstation is used to draw the initial image (Frame f). If this operation fails, the error handler is invoked. Otherwise the valuater is read (readValuator) to obtain the scalar (real) value supplied by the user. Using the initial Frame and the user-supplied scalar, the function interactiveScale is invoked to repeatedly perform the following actions:

   1. scale the given Frame by the specified scalar value
   2. display the scaled Frame
   3. read the valuater device to obtain the new scalar x
   4. If x is 0.0 terminate the list of I/O requests (thereby terminating the program), otherwise invoke itself with the scaled Frame and x as parameters.

   window and viewport are passed to the showFrame command.

   The graphical output generated by this program is shown in Figure B-1.
2. Program to Demonstrate the "Rubberbanding" Operation

Moving one end of a line segment while the other end of the segment is kept stationary is a trivial animation sequence and is frequently used in graphics packages that allow the user to draw figures on the screen. This is traditionally known as "rubberbanding". The following Haskell program demonstrates how this feature can be implemented with ease using the FunGraf framework. The program introduces to the user the use of the following features of FunGraf:

* the stroke device
* continuation I/O style for
  - implementing repetetive user-interface actions (while loops)
  - creating an illusion of moving objects using graphics writing modes (trivial form of animation).
* viewport to window transformation

Evaluation of the program begins with function *main*, which opens the workstation identified by the identifier *workstation*. If an error is encountered in opening the workstation, the error handler *errorHandler* prints the name of the error condition and aborts. If the workstation is successfully opened, the channel identification associated with the workstation is used to wait for the user to press the locator button. If the point is read in successfully, the line joining the anchor point (Pt2 0.0 0.0) and the input point $p$ is encapsulated in a Frame and the Frame is displayed. Note that the graphics drawing attributes supplied with the *showFrame* call indicate that the drawing is done in the XOR mode. If this operation fails, the error handler is invoked. If not, the function *trackCursor* is invoked with the channel *channel* and the Frame as its arguments. *trackCursor* performs the following actions repeatedly:

1. read the stroke device to read the next position of the locator (the mouse):

   a) If the point *new_point* is read successfully, the argument $fr$ is redrawn in the XOR drawing mode. Since drawing the same image twice in XOR mode erases the image, the old Frame is erased. Then a new Frame is formed containing the line joining the anchor point and the new point *new_point*. The new frame is displayed in the XOR mode.
At this point, once cycle of operations is complete. The function \textit{trackCursor} invokes itself recursively using the arguments \textit{channel} and the newly formed \textit{Frame}.

b) Should the user release the locator button, the device handler sends an appropriate response and the Haskell interpreter invokes the failure continuation. This failure continuation is a normal termination of the input from the stroke device. When this continuation is taken, the program terminates the list of requests to the Haskell system, thereby terminating the execution of the program.

This program uses the function \textit{viewportToWindow} and thereby illustrates the simple functional implementation of one of the most useful computations in graphics: window-to-window linear transformation.

The graphical output generated by this program is shown in Figure B-2.
3. Program to demonstrate the use of String Input Device

The keyboard device is not as often used in interactive computer graphics as is the locator device. However, inputting, manipulating and displaying character strings is a required task and the following program illustrates how the string input device may be used interactively for obtaining input in a graphics program. This program reads a character string and displays it at every point clicked upon on the screen by the user.

Specifically, the program introduces the user following:

- reading input from the String Input Device in continuation style
- displaying character strings interactively

The program opens the named workstation (workstationName) and if successful, waits on the channel just opened (channel) for the user to type in a character string. The RETURN character is recognized as the termination of the input character sequence. Should there be any error in this process, the error handler errorHandler is invoked which prints out the appropriate diagnostic message. Usually the only error detected by the system in the string input request is the specification of an unopened channel. If the input is successful, the success continuation is invoked which is a function that displays the character string just read in on the channel channel. In order to display any graphical object, it must be encapsulated in a Frame. Hence the character string is put into a Frame frame and displayed. If this step is successful, the cursor tracking function trackMouse is invoked with the channel identifier channel and the string stry as its arguments. Function trackMouse reads the locator on the channel channel, converts it to the coordinates of the world and displays the Frame containing the character string at that point.

The graphical output generated by this program is shown in Figure B-3.
4. Intensive user interaction: "paintbrush" utility

The ability to continuously keep track of user generated events on a particular channel and to generate appropriate reactions to those events requires a high degree of user interaction capability on the part of the programming system. Such a capability of the FunGraf system is illustrated using the "paintbrush" program described in this section. "Painting" the display surface using the locator (mouse) cursor is a feature that is used in many simple and complex image synthesis packages. Effectively, the position of the locator cursor is continuously tracked and points are plotted at the detected cursor positions.

The program introduces the following:

- the stroke device in the continuation I/O style
- writing a simple geometric membership check function to check if the specified point is a member of the specified window (a commonly used graphics function).

The program opens the specified workstation (workstationName) and if this succeeds, draws the point (Pt2 30.0 30.0) in world coordinates on the display surface. Once this has been done successfully, the function trackMouse is invoked with the channel identifier channel as its argument. Function trackMouse reads the next point input by the user using the stroke device (input function readStroke). If this read is successful, the resultant point is converted into world coordinates, packaged into a Frame and displayed. If, however, the read from the Stroke device is unsuccessful, the program recognizes it to be the termination signal from the user and terminates the request sequence by the command done. Once the request sequence terminates, the program execution terminates.

This program uses the function point2InWindow that checks the membership of the given two-dimensional point to the specified window. While this is a trivial function, its generic form is used quite often in graphics.

The graphical output generated by this program is shown in Figure B-4.
5. A simple menu-based landscape synthesis package

This is the most complex application that we built using the FunGraf framework. This application uses most features discussed above and combines them to arrive at an application that may be used as a template to write more useful applications.

The program provides a workstation with the following devices: locator, string input, pick, display and the valuator. The user is provided an empty "canvas" to start with. The user may incrementally build a complex picture on this canvas using the devices. A "notice board" is displayed which continuously shows context sensitive help instructions and the results of the user's last action. The canvas has a menubar with the following menu options: Add, Delete, Help and Exit. The Add option causes a menu screen to come up from which the user may select primitive picture segments to add to the canvas. The Delete option erases the picture segment on the canvas identified by the user. The Help button displays instructions on the notice board. The Exit button causes the program to terminate.

The Add option leads to a menu of simple and composite picture segments any of which may be selected and optionally subjected to rotation or scaling before adding the segment to the canvas. The menu (Figure B-5 (a)) displays icons of the following types: (a) houses (b) trees (c) lamp poles (d) vehicles and (e) composite. Clicking on the house menu shows on the menu window a list of different types of houses (Figure B-5 (b)). Similarly, there exist menu of trees, lamp poles and vehicles. After an object has been selected from the menu, it can be transformed (rotation/scaling) (Figure B-5 (c)) before it is placed on the canvas. The composite menu option provides a way for the user to create new primitive segments. Selecting the composite menu option on the segments menu brings up a secondary canvas upon which the user may define an arbitrary figure (Figure B-5 (e)) using the lower level primitives, namely, points, lines, circles, text strings and paints (streams of points).

The point option allows the user to place a point at the desired place on the canvas. If this option is selected, the user is asked to click on a point on the canvas and a point is plotted at the spot. The line option allows the user to place a line on the canvas. The line is identified by the beginning point and the ending point, both of which the user identifies using the locator. Using the circle option, the user can place on the canvas circles of varying sizes. The user is expected to click on the centre of the circle and at any point on the circumference of the circle. The text option can be used to place a string of characters at the specified point on the canvas. If this option can be used to place a string of characters at the specified point on the canvas. If this option is selected, the program waits for the user to identify (click on) the position of the
string on the canvas and subsequently waits for input on the string input device. The composite segment too can be transformed before it is placed on the canvas.

The Figures B-5 (a) through B-5 (h) show a sample execution of the MENU.HAS program starting from an empty canvas (a) and arriving finally at the complete picture (h).
module Scale where
import PreludeRealCore
import PreludeMy

-- MAIN MODULE : WHERE EVALUATION BEGINS
main :: Dialogue
main = openWorkStation workstationName [(ValuatorNeeded 1)] abort
   \channel -> showFrame channel frame window viewport bundle
       errorHandler
       (readValuator channel errorHandler
       (interactiveScale channel)
     )

-- ADDRESS OF THE WORKSTATION
workstationName :: String
workstationName = "nms.his.com:0"

-- WORKSTATION ATTRIBUTE TO INDICATE THAT ANY DRAWING MUST BE
-- PRECEDED BY CLEARING THE DISPLAY SURFACE.
bundle :: GraphicsContext
bundle = [(ClearBeforeDump True)]

-- WINDOW AND VIEWPORT DEFINITIONS :
window, viewport :: Window2
window = (MkWindow2 (Pt2 0.0 0.0) (Pt2 100.0 100.0))
viewport = (MkWindow2 (Pt2 0.0 0.0) (Pt2 1.0 1.0))

-- THE INITIAL FRAME (IMAGE)
frame :: Frame
frame = (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 0.0)
                                       (Pt2 50.0 50.0))
                    (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 50.0)
                                                  (Pt2 50.0 0.0))
                            (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 0.0)
                                                      (Pt2 0.0 0.0))
                                      )
                                  )
                            )
                     )
                )
            )
       )
     )
   )
-- FUNCTIONS TO HANDLE I/O ERRORS :
errorHandler :: IOError -> Dialogue
errorHandler errMsg = appendChan stdout (extractString errMsg) abort

done

extractString :: IOError -> String
extractString (OtherError str) = str
extractString _ = "Std Error"
-- THIS PROGRAM DEMONSTRATES HOW TO IMPLEMENT RUBBERBANDING USING THE
-- FUNCTIONAL GRAPHICS SYSTEM. THE DEVICES USED: THE LOCATOR AND THE
-- STROKE DEVICE.

module RubberBand where
import PreludeRealCore
import PreludeMy

-- MAIN MODULE IS WHERE THE EVALUATION BEGINS
main :: Dialogue
main = openWorkStation workstationName ws_bundle abort
   (\channel ->
    readLocator channel CUR_PICK
    errorHandler
    (\ptx ->
     showFrame channel fr window viewport bundle
     errorHandler
     (trackCursor channel fr)
     where
      fr = MkFrame (Line2ToVector2
                      (MkLine2 (Pt2 0.0 0.0) sel pt))
             EmptyFrame ;
      sel pt = viewportToWorld ptx viewport window
   )
   )

-- MODULE THAT INTERACTS WITH THE USER TO MONITOR THE CURRENT
-- CURSOR POSITION AND TO ERASE THE LAST LINE AND TO DRAW THE
-- NEW "STRETCHED" LINE:;
trackCursor :: Channel -> Frame -> Dialogue
trackCursor channel frame =
   readStroke channel
   (\errx -> appendChan stdout "Done rubber" abort done)
   (\ptnew -> showFrame channel newfr window viewport bundle
    errorHandler
    (showFrame channel frame window viewport bundle
     errorHandler -- ERROR: FAILED TO DISPLAY
     -- THE STRETCHED LINE.
     (trackCursor channel newfr)
     -- OK; ONTO NEXT ITERATION.
    )
    where
      newfr = MkFrame (Line2ToVector2
                        (MkLine2 (Pt2 0.0 0.0) newpt))
                        EmptyFrame ;
      newpt = viewportToWorld ptnew viewport window
   )

-- ROUTINE TO DETECT IF THE GIVEN POINT LIES IN THE SPECIFIED
-- WINDOW :;
point2InWindow2 :: Window2 -> Point2 -> Bool
point2InWindow2 (MkWindow2 (Pt2 xl yl) (Pt2 xh yh)) (Pt2 x y) =
   ((xl <= x) && (x <= xh) && (yl <= y) && (y <= yh))
-- TRANSFORM THE GIVEN POINT FROM ONE WINDOW TO THE OTHER ::
viewportToWindow :: Window2 -> Window2 -> Point2 -> Point2
viewportToWindow (MkWindow2 (Pt2 x11 y11) (Pt2 xh1 yh1))
    (MkWindow2 (Pt2 x12 y12) (Pt2 xh2 yh2)) (Pt2 x y)
    = (Pt2 xx yy)
where {
    xx = xaspect_ratio*(x - x11) + x12 ;
    yy = yaspect_ratio*(y - y11) + y12 ;
    xaspect_ratio = ((xh2 - x12)/(xh1 - x11));
    yaspect_ratio = ((yh2 - y12)/(yh1 - y11))
}

-- A SPECIAL CASE OF THE LAST FUNCTION: MAP POINTS FROM VIEWPORT
-- TO THE WORLD WINDOW.
viewportToWorld :: Point2 -> Window2 -> Window2 -> Point2
viewportToWorld point v w =
    if (point2inWindow2 v point) then
        viewportToWorld v w point
    else
        EmptyPoint?

-- ATTRIBUTES WITH WHICH TO OPEN THE WINDOW
ws_bundle :: BundledAttributes
ws_bundle = [(ValuatorNeeded 0)]

bundle :: GraphicsContext
bundle = [(WriteMode MXcr)]

workstationName :: String
workstationName = "nms.hls.com:0"

window, viewport :: Window2
window = (MkWindow2 (Pt2 0.0 0.0) (Pt2 100.0 100.0))
viewport = (MkWindow2 (Pt2 0.0 0.0) (Pt2 1.0 1.0))

-- I/O ERROR HANDLER
errorHandler :: I0Error -> Dialogue
errorHandler errMsg = appendChan stdout (extractString errMsg)
    abort
    done

extractString :: I0Error -> String
extractString (OtherError str) = str
extractString _ = "Std Error"

112
-- THE FOLLOWING PROGRAM ILLUSTRATES THE USE OF THE Keyboard DEVICE
-- AND THE LOCATOR DEVICE :: IT ALSO ILLUSTRATES THE IMPLEMENTATION
-- OF USER INTERACTION USING FUNCTIONAL INPUT/OUTPUT.

module StringDevice where
import PreludeRealCore
import PreludeMy

-- EVALUATION BEGINS HERE:
main :: Dialogue
main = openWorkStation workstationName ws_bundle abort
  -- OPENED WORKSTATION; READ A STRING NOW ::
  (\channel ->
    readStringDevice channel
    errorHandler -- COULDN'T READ STRING !!
    (\stry -> showFrame channel frame window viewport
      bundle
      errorHandler
      (trackMouse channel stry)
      where
        frame = MkFrame (LabelToVector2 label1) EmptyFrame
        label1 = MkLabel stry (Pt2 30.0 30.0)
        window viewport
    )
  )

-- MODULE THAT KEEPS TRACK OF CURSOR POSITION AND DISPLAYS THE
-- TYPED IN STRING AT THE CURSOR POSITION.
trackMouse :: Channel -> String -> Dialogue
trackMouse channel strc =
  readLocator channel CUR_PLACE
  errorHandler
  (\ptnew -> showFrame channel newfr window viewport
    bundle
    errorHandler
    (trackMouse channel strc)
    where
      newfr = MkFrame (LabelToVector2
        (MkLabel strc newpt window viewport)
      ) EmptyFrame
      newpt = viewportToWorld ptnew viewport window
  )

-- MODULE THAT INDICATES IF A GIVEN POINT LIES IN THE GIVEN WINDOW
point2InWindow2 :: Window2 -> Point2 -> Bool
point2InWindow2 (MkWindow2 (Pt2 x1 y1) (Pt2 xh yh)) (Pt2 x y) =
  ((x1 <= x) && (x <= xh) && (y1 <= y) && (y <= yh))

-- TRANSFORM THE GIVEN POINT FROM ONE WINDOW THE OTHER
viewportToWindow :: Window2 -> Window2 -> Point2 -> Point2
viewportToWindow (MkWindow2 (Pt2 x1 y11) (Pt2 xh1 yh1))
  (MkWindow2 (Pt2 x12 y12) (Pt2 xh2 yh2))
  (Pt2 x y) =
  (Pt2 xx yy)
where 

    xx = xaspect_ratio*(x - x11) + x12 ;
    yy = yaspect_ratio*(y - y11) + y12 ;
    xaspect_ratio = ((xh2 - x12)/(xh1 - x11));
    yaspect_ratio = ((yh2 - y12)/(yh1 - y11))

-- FUNCTION TO MAP A POINT FROM THE VIEWPORT TO THE WORLD 
-- (USING PREVIOUS FUNCTION)
viewportToWorld :: Point2 -> Window2 -> Point2
viewportToWorld point v w =
    if (point2InWindow2 v point) then
        viewportToWindow v w point
    else
        EmptyPoint2

workstationName :: String
workstationName = "nms.hls.com:0"

ws_bundle :: BundledAttributes
ws_bundle = [(ValuatorNeeded 1)]

bundle :: GraphicsContext
bundle = [(WriteMode MCcopy)]

window, viewport :: Window2
window = (MkWindow2 (Pt2 0.0 0.0) (Pt2 100.0 100.0))
viewport = (MkWindow2 (Pt2 0.0 0.0) (Pt2 1.0 1.0))

-- I/O ERROR HANDLER ::
errorHandler :: IOError -> Dialogue
errorHandler errMsg = appendChan stdout (extractString errMsg)
    abort
    done

extractString :: IOError -> String
extractString (OtherError str) = str
extractString _ = "Std Error"
-- THIS PROGRAM DEMONSTRATES THE IMPLEMENTATION OF THE PAINTBRUSH
-- UTILITY USING THE FUNCTIONAL GRAPHICS SYSTEM. THE CHIEF DEVICE
-- USED IS THE STROKE DEVICE.

module PaintBrush where
import PreludeRealCore
import PreludeMy

-- MAIN (DRIVER) FUNCTION ::
main :: Dialogue
main = openWorkStation workstationName ws_bundle
   abort
   (\channel -> showFrame channel frame window viewport bundle
      errorHandler
      (trackMouse channel)
      where
         frame = MkFrame (Pt2ToVector2 (Pt2 30.0 30.0))
             EmptyFrame
   )

-- FUNCTION THAT KEEPS TRACK OF THE CURSOR POSITION AND PLOTS POINT
-- AT LAST CURSOR POSITION, GIVING THE ILLUSION OF PAINTING.
trackMouse :: Channel -> Dialogue
trackMouse channel = readStroke channel
   (\error -> closeWorkStation channel abort done)
   (\ptnew -> showFrame channel newfr window
      viewport bundle
      errorHandler
      (trackMouse channel)
      where
         newfr = MkFrame (Pt2ToVector2 newpt)
             EmptyFrame
         newpt = viewportToWorld ptnew
             viewport window
   )

-- FUNCTION TO TEST IF THE GIVEN POINT LIES IN THE GIVEN WINDOW ::
point2InWindow2 :: Window2 -> Point2 -> Bool
point2InWindow2 (MkWindow2 (Pt2 x1 y1) (Pt2 xh yh)) (Pt2 x y)
   = ((x1 <= x) && (x <= xh) && (y1 <= y) && (y <= yh))

-- FUNCTION TO MAP A POINT FROM ONE WINDOW TO THE OTHER ::
viewportToWindow :: Window2 -> Window2 -> Point2 -> Point2
viewportToWindow (MkWindow2 (Pt2 x11 y11) (Pt2 xh1 yh1))
   (MkWindow2 (Pt2 x12 y12) (Pt2 xh2 yh2))
   (Pt2 x y) =
   (Pt2 xx yy)
   where {
      xx = xaspect_ratio*(x - x11) + x12 ;
      yy = yaspect_ratio*(y - y11) + y12 ;
      xaspect_ratio = ((xh2 - x12)/(xh1 - x11));
      yaspect_ratio = ((yh2 - y12)/(yh1 - y11))
   }

-- MAP THE POINT FROM VIEWPORT TO THE WORLD ::
viewportToWorld :: Point2 -> Window2 -> Window2 -> Point2
viewportToWorld point v w =
  if (point2InWindow2 v point) then
    viewportToWindow v w point
  else
    EmptyPoint2

workstationName :: String
workstationName = "nms.hls.com:0"

ws_bundle :: BundledAttributes
ws_bundle = [(ValuatorNeeded 0)]

bundle :: GraphicsContext
bundle = [(WriteMode MCopy)]

window :: Window2
window = (MkWindow2 (Pt2 0.0 0.0) (Pt2 100.0 100.0))
viewport = (MkWindow2 (Pt2 0.0 0.0) (Pt2 1.0 1.0))

-- I/O ERROR HANDLER ::
errorHandler :: IOError -> Dialogue
errorHandler errMsg = appendChan stdout (extractString errMsg)
  abort
  done

extractString :: IOPermissionError -> String
extractString (OtherError str) = str
extractString _ = "Std Error"
Figure B-5(b): Menu of Picture Segments
Figure B-5(c): User Selects a house from the House Menu
Figure B-5(d): User is asked to select a pole from the Pole Menu
Figure B-5(f): Creation of a composite segment using the Composite Menu
module Keyboard where
import PreludeRealCore
import PreludeMy

-- TYPE PSEUDONYMS :: WILL BE USED FREQUENTLY USED IN THIS PROGRAM::
-- **************** Type Definitions ****************
type Transform = Channel -> Channel -> Channel -> Frame -> Dialogue
type GeometricTransform = Transform -> Channel -> Channel -> Channel
-> Frame -> Frame -> Dialogue
type CompositeTransform = Transform -> Channel -> Channel
-> Channel -> Frame -> Frame -> Dialogue
type CommandDecode = String -> Int
-- **************** Type Definitions ****************

-- MODULE THAT BEGINS EVALUATION: OPENS ALL THE CHANNELS AND INVOKES
-- THE MENU CONTROLLER.
main :: Dialogue
main = openWorkStation workstationName ws_bundle
  -- OPEN THE "CANVAS" CHANNEL
  errorHandler
    (\channel -> if (channel >= 0) then
      (openWorkStation workstationName
        (ValuatorNeeded 0):{WsWidth 500}:
          {WHeight 500}:{WsName "Menu"}[]):})
    -- OPEN THE NOTICE BOARD CHANNEL
  errorHandler
    (\menu ->
    openWorkStation workstationName
    notice_bundle
    -- OPEN THE MENU CHANNEL
    errorHandler
      (\notice_board ->
        showFrame notice_board (noticeFunction
          initial_mess) notice_window viewprt
        bundle
        errorHandler
        (main_menu_control channel
          notice_board menu EmptyFrame)
      )
    )
  )
else
  abnormalExit

-- MAIN MENU CONTROLLER: PUTS UP THE MENU ON THE CANVAS CHANNEL AND
-- INTERPRETS THE USER COMMANDS AND INVOKES THE APPROPRIATE FUNCTIONS
main_menu_control :: Transform
main_menu_control channel notice_board menu curr_fr =
  showFrame channel main_panel windowx viewprt bundle
  errorHandler
(showFrame channel curr_fr windowx viewprt [])
errorHandler
(pickFrame channel CUR_PICK windowx viewprt [main_panel]
errorHandler
\sel_frame -> case command of
     -- ILLEGAL COMMAND WAS SELECTED BY USER: WARN AND
     -- REPEAT ACTIONS
  0 -> showFrame notice_board (noticeFunction:
      nobutton_err_mess) notice_window viewprt
      bundle
      errorHandler
      (main_menu_control channel notice_board
      menu curr_fr)
     -- ADD PICTURE
  1 -> addPic curr_fr main_menu_control channel
      notice_board menu
     -- ERASE PICTURE
  2 -> deletePic curr_fr main_menu_control channel
      notice_board menu
     -- DISPLAY HELP MESSAGE ON NOTICE BOARD
  3 -> showFrame notice_board (noticeFunction:
      main_help_mess) notice_window viewprt bundle
      errorHandler
      (main_menu_control channel notice_board
      menu curr_fr)
     -- USER OPTED TO EXIT: CLOSE ALL CHANNELS AND EXIT
  4 -> showFrame notice_board
      (noticeFunction exit_mess) notice_window
      viewprt bundle
      errorHandler
      (closeWorkStation channel errorHandler
      (closeWorkStation notice_board errorHandler
      (closeWorkStation menu errorHandler done)
      )
    )
  )

  )

where -- MOUSE SELECTION TO COMMAND DECODER
  command = button_to_command
    (label_string sel_frame)
  )

)
if (nullFrame sel_frame) then
   addPic curr_fr func channel notice_board menu
else
   (case sel_icon of
      -- ILLEGAL OR NO SELECTION: REPEAT
      0 -> addPic curr_fr func channel notice_board menu
      -- ADD A HOUSE SEGMENT TO THE CANVAS
      1 -> showFrame notice_board (noticeFunction
         add_mess) notice_window viewprt bundle
         errorHandler
         (showFrame menu house_menu menu_window viewprt
            bundle
            errorHandler
            (pickFrame menu CUR_PICK menu_window viewprt
               [house_menu]
               errorHandler
               (\sel_house ->
                     geometric_transform func channel
                     notice_board menu curr_fr
                     sel_house
               )
            )
         )
      )
      -- ADD A TREE SEGMENT TO THE CANVAS
      2 -> showFrame notice_board (noticeFunction
         add_mess) notice_window viewprt bundle
         errorHandler
         (showFrame menu tree_menu menu_window viewprt
            bundle
            errorHandler
            (pickFrame menu CUR_PICK menu_window viewprt
               [tree_menu]
               errorHandler
               (\sel_tree ->
                   geometric_transform func channel
                   notice_board menu curr_fr sel_tree
               )
            )
         )
      )
      -- ADD A POLE SEGMENT TO THE CANVAS
      3 -> showFrame notice_board (noticeFunction
         add_mess) notice_window viewprt bundle
         errorHandler
         (showFrame menu pole_menu menu_window viewprt
            bundle
            errorHandler
            (pickFrame menu CUR_PICK menu_window
               viewprt [pole_menu]
               errorHandler
               (\sel_pole ->
                   geometric_transform func channel
                   notice_board menu curr_fr sel_pole
               )
            )
         )
      )
      -- ADD A COMPOSITE SEGMENT TO THE CANVAS

129
4 -> showFrame notice_board (noticeFunction
   ADD_MESS) notice_window viewprt bundle
   errorHandler
   (openWorkStation workstationName
     (ValuatorNeeded 0)::BundledAttributes)
   errorHandler
     (\make_board ->
     make_composite_frame func channel
     notice_board menu make_board
     curr_fr EmptyFrame
   )

5 -> showFrame notice_board (noticeFunction
   add_mess) notice_window viewprt bundle
   errorHandler
   (showFrame menu car_menu menu_window viewprt
     bundle
     errorHandler
     (pickFrame menu CUR_PICK menu_window
      viewprt [car_menu]
     errorHandler
       (\sel_car ->
        geometric_transform func channel
        notice_board menu curr_fr sel_car
       )
     )
   )
where -- DECODE THE USER COMMAND
   sel_icon = get_selected_label sel_frame
)
)
)
)
)
)

-- MODULE TO ERASE A PICTURE FROM THE CANVAS:: CANVAS IS INITIALLY
-- AN EMPTY FRAME. FRAMES ARE ADDED TO IT. THE FRAMES MAY THEN BE
-- ERASED USING THIS MODULE
deletePic :: Frame -> Transform -> Channel -> Channel ->
   Channel ->
   Dialogue
deletePic curr_fr func channel notice_board menu
   | nullFrame curr_fr =
   | showFrame notice_board (noticeFunction del_err_mess)
   | notice_window viewprt bundle
   | errorHandler
   | (func channel notice_board menu EmptyFrame)
   | otherwise =
   | (showFrame notice_board (noticeFunction delete_mess)
   | notice_window viewprt bundle
   | errorHandler
   | (pickFrame channel CUR_DANGER windowx viewprt [curr_fr]
   | errorHandler
   | (\sel_frame ->
   | (case sel_frame of
   | EmptyFrame ->
   | showFrame notice_board (noticeFunction

130
```c
no_del_mess) notice_window viewprt bundle
    errorHandler
    (func channel notice_board menu curr_fr)

(MkFrame v f) ->
    showFrame notice_board (noticeFunction
delete_done_mess) notice_window viewprt
    bundle
    errorHandler
    (showFrame menu (MkFrame v f) menu_window
    viewprt ((ClearBeforeDump True):[])
    errorHandler
    (func channel notice_board menu
    (filterFrame curr_fr (MkFrame v f)))

)

)


-- MODULE TO PERFORM A GEOMETRIC TRANSFORM ON THE GIVEN FRAME
-- (TRANSLATION/ROTATION/SCALING). THE FRAME TO BE TRANSFORMED IS THE
-- LAST ARGUMENT, THE SECOND LAST ARGUMENT IS THE CUMULATIVE FRAME
-- INTO WHICH THE TRANSFORMED FRAME MUST BE ADDED AND THE MAIN LOOP
-- REINVOKED.

geometric_transform :: GeometricTransform
geometric_transform func channel notice_board menu curr_frame
    new_frame =
    showFrame notice_board (noticeFunction geom_mess) notice_window
    viewprt bundle
    errorHandler
    (showFrame channel (UnionFrame geometry_panel new_frame)
    windowx viewprt bundle
    errorHandler
    (pickFrame channel CUR_PICK windowx viewprt [geometry_panel]
    errorHandler
    (\sel_frame ->
    case command of
        -- ILLEGAL OR NO SELECTION : REPEAT
        0 -> geometric_transform func channel notice_board
            menu curr_frame new_frame
        -- SCALE THE FRAME ABOUT THE PIVOT USING THE
        -- VALUATOR OUTPUT
        1 -> showFrame notice_board (noticeFunction
            scale_pivot_mess) notice_window
            viewprt bundle
            errorHandler
            (readLocator channel CUR_CIRCLE
            errorHandler
            (\ptx ->
            case ptx of
                EmptyPoint2 ->
                    fatalErrorHandler
                    "Scale Pivot cannot be EmptyPoint2"
                (Pt2 x y) ->
                    (readValuator channel
```
errorHandler
    (\fac ->
        geometric_transform func channel
        channel notice_board menu
        curr_frame scaled_frame
        where
        scaled_frame =
            moveFrame dscaled_frame pivot_pt
dscaled_frame =
            scaleFrame nnew_frame (10.0*fac)
nnew_frame =
            displaceBack new_frame pivot_pt
        pivot_pt = viewportToWorld
            (Pt2 x y) viewpt
            windowx
    )
)
)
-- ROTATE THE FRAME ABOUT THE PIVOT USING THE
-- VALUATOR OUTPUT
  2 -> showFrame notice_board (noticeFunction
      rotate_pivot_mess) notice_window viewpt
      bundle
      errorHandler
      (readLocator channel CUR_CIRCLE
      errorHandler
        (\ptx ->
            case ptx of
                EmptyPoint2 ->
                    fatalErrorHandler
                        "EmptyPoint2 as Pivot"
                (Pt2 x y) ->
                    readValuator channel
                    errorHandler
                    (\fac ->
                        geometric_transform func
                        channel notice_board
                        menu curr_frame
                        rotated_frame
                        where
                        rotated_frame =
                            rotateFrame new_frame
                            pivot_pt (fac*360.0)
pivot_pt = viewportToWorld
                (Pt2 x y) viewpt
                windowx
            )
        )
      )
    )
-- DISPLAY HELP MESSAGE
  3 -> showFrame notice_board (noticeFunction
      geom_help_mess) notice_window viewpt
      bundle
      errorHandler
      (geometric_transform func channel

132
notice_board menu curr_frame new_frame

-- END GEOMETRIC TRANSFORMATIONS: PLACE THE
-- TRANSFORMED FRAME IN THE CUMULATIVE FRAME AND
-- RETURN TO MAIN MENU
4 -> showFrame notice_board (noticeFunction
  geom_done_mess) notice_window viewprt
  bundle
errorHandler
 (placeFunction func channel notice_board
  menu curr_frame new_frame)
  where
  command = geom_button_to_command (label_string
  sel_frame)
)
)

-- MODULE TO TAKE USER INPUT TO PLACE THE GIVEN FRAME ON THE CANVAS
placeFunction :: Transform -> Channel -> Channel -> Channel ->
Frame -> Frame -> Dialogue
placeFunction func channel notice_board menu curr_frame new_frame =
  showFrame notice_board (noticeFunction place_mess)
  notice_window viewprt bundle
errorHandler
  [showFrame channel curr_frame windowx viewprt bundle
errorHandler
     (readLocator channel CUR_PLACE
      errorHandler
     (\pt -> placeFrame func channel notice_board menu
      curr_frame new_frame pt
     ))
  ]
)

placeFrame :: Transform -> Channel -> Channel -> Channel ->
Frame -> Frame -> Point2 -> Dialogue
placeFrame func channel notice_board menu curr_frame new_frame

  EmptyPoint2 = _
closeWorkStation channel errorHandler
 (closeWorkStation notice_board errorHandler
 (closeWorkStation menu errorHandler
 (fatalErrorHandler
   "[Fatal :: Received EmptyPoint2 for Frame Placement]"
 ))
 )
)

placeFrame func channel notice_board menu EmptyFrame new_frame

(Pt2 x y) =

func channel notice_board menu positioned_frame
where
  positioned_frame = moveFrame new_framexy position_point;
  position_point = viewportToWorld (Pt2 x y) viewprt windowx;
  new_framexy = subtractXComponent new_framex (least_xy 2
  new_frame 10000000.0) ;
  new_framex = subtractXComponent new_frame (least_xy 1
new_frame 10000000.0)

placeFrame func channel notice_board menu curr_frame
    new_frame (Pt2 x y) =
    func channel notice_board menu (UnionFrame curr_frame
        positioned_frame)

    where
    positioned_frame = moveFrame new_frame xy position_point;
    position_point = viewportToWorld (Pt2 x y) viewprt windowx;
    new_frame xy = subtractYcomponent new_frame (least xy 2
        new_frame 10000000.0) ;
    new_frame xy = subtractXcomponent new_frame (least xy 1
        new_frame 10000000.0)

    -- MODULE TO CONSTRUCT A COMPOSITE FRAME TO BE PLACED INTO
    -- THE CANVAS. THE PRIMITIVES POINT, LINE, CIRCLE AND TEXT
    -- CAN BE USED TO CONSTRUCT SUCH A SEGMENT.
    makeCompositeFrame func channel notice_board menu makeBoard
        curr_fr created_frame =
        showFrame make_board make_panel windowx viewprt []
        errorHandler
        (showFrame make_board created_frame windowx viewprt []
        errorHandler
        (pickFrame make_board CUR_PICK windowx viewprt
            [make_panel]
        errorHandler
        (sel_frame ->
        case command of
        -- ILLEGAL OR NO SELECTION: SHOW ERROR MESSAGE
        0 -> showFrame notice_board (noticeFunction
            make_err_msg) notice_window viewprt
            bundle
        errorHandler

            (makeCompositeFrame func channel
                notice_board menu make_board curr_fr
                created_frame)
        -- ADD A POINT TO THE SEGMENT
        1 -> addPoint curr_fr created_frame func channel
            notice_board menu make_board
        -- ADD A LINE TO THE SEGMENT
        2 -> addLine curr_fr created_frame func channel
            notice_board menu make_board
        -- ADD A CIRCLE TO THE SEGMENT
        3 -> addCircle curr_fr created_frame func channel
            notice_board menu make_board
        -- ADD A TEXT TO THE SEGMENT
        4 -> addText curr_fr created_frame func channel
            notice_board menu make_board
        -- FINISHED BUILDING THE SEGMENT: PLACE IT ON
        -- THE CANVAS AND CLOSE THE COMPOSITE CHANNEL
        5 -> showFrame notice_board (noticeFunction
            make_done_msg) notice_window viewprt
            bundle
        errorHandler
        (closeWorkStation make_board
errorHandler
  (geometric_transform func channel
    notice_board menu curr_fr created_frame
  )
)
where
  command = make_button_to_command
    (label_string sel_frame)
)

-- MODULE TO ADD A USER SPECIFIED POINT TO THE CUMULATIVE FRAME
addPoint :: Frame -> Frame -> Transform -> Channel -> Channel
            -> Channel -> Channel -> Dialogue
addPoint curr_fr created_frame func channel notice_board
  menu make_board =
    showFrame notice_board (noticeFunction make_point_mess)
      notice_window viewprt bundle
      errorHandler
      (readLocator make_board CUR_PENCIL
        errorHandler
        (\ptx ->
        case ptx of
          EmptyPoint2 ->
            fatalError
            "[Fatal :: Attempt to include EmptyPoint2 in Frame]"
          (Pt2 x y) ->
            make_composite_frame func channel notice_board
              menu make_board curr_fr
            (mergeVectorInNormalFrame created_frame
              (Pt2ToVector2 point)
            )
        )
      )
    )

-- MODULE TO ADD A USER SPECIFIED LINE TO THE CUMULATIVE FRAME
addLine :: Frame -> Frame -> Transform -> Channel -> Channel
                 -> Channel -> Channel -> Dialogue
addLine curr_fr created_frame func channel notice_board
  menu make_board =
    showFrame notice_board (noticeFunction make_line_mess)
      notice_window viewprt bundle
      errorHandler
      (readLocator make_board CUR_CROSS
        errorHandler
        (\startpt ->
        showFrame notice_board (noticeFunction
          make_line_end_mess) notice_window viewprt bundle
          errorHandler
          (readLocator make_board CUR_PENCIL
            errorHandler
            (\endpt ->
            case endpt of
              EmptyPoint2 ->
            )
            )
            fatalError
            "[Fatal :: Attempt to include EmptyPoint2 in Frame]"
          )
        )
      )
"[Fatal :: Attempt to include EmptyPoint2 in Frame]"

(Pt2 x y) ->
showFrame notice_board (noticeFunction
done_line_mess) notice_window viewprt
bundle
errorHandler
    (make_composite_frame func channel
     notice_board
     menu make_board curr_fr
     (mergeVectorInNormalFrame created_frame line)
     where
     line = Line2ToVector2
     (MkLine2 start end);
     start = viewportToWorld startpt viewprt
     windowx;
     end = viewportToWorld (Pt2 x y) viewprt
     windowx
    )
)
)
)

-- MODULE TO ADD A USER SPECIFIED CIRCLE TO THE
-- CUMULATIVE FRAME
addCircle :: Frame -> Frame -> Transform -> Channel ->
    Channel -> Channel -> Channel -> Dialogue
addCircle curr_fr created_frame func channel notice_board
    menu make_board =
    showFrame notice_board (noticeFunction make_circle_mess)
    notice_window viewprt bundle
errorHandler
    (readLocator make_board CUR_CROSS
     errorHandler
    (\startpt ->
     showFrame notice_board (noticeFunction
     make_radius_mess) notice_window viewprt bundle
     errorHandler
     (readLocator make_board CUR_PENCIL
      errorHandler
    (\endpt ->
     showFrame notice_board (noticeFunction
     done_circle_mess) notice_window viewprt bundle
     errorHandler
     (case endpt of
     EmptyPoint2 ->
     fatalErrorHandler
     "[Fatal :: Attempt to include EmptyPoint2 in Circle]"
     (Pt2 x y) ->
     showFrame notice_board (noticeFunction
     done_line_mess) notice_window
     viewprt bundle
     errorHandler
     (make_composite_frame func channel
     notice_board menu make_board curr_fr
(mergeVectorInNormalFrame
  created_frame circlex)
where
circlex =
  Arc2ToVector2 (MkArc2 centrex radiusx
    0.0 360.0);
radiusx = distancex centrex endx
centrex = viewportToWorld startpt
  viewprt
  windowx;
endx = viewportToWorld endpt viewprt
  windowx
)
)
)
)

-- MODULE TO A ADD USER SPECIFIED TEXT STRING TO THE
-- CUMULATIVE FRAME :
addText :: Frame -> Frame -> Transform -> Channel ->
  Channel -> Channel -> Channel -> Dialogue
addText curr_fr created_frame func channel notice_board
  menu make_board =
(readStringDevice make_board
  errorHandler
  \strx ->
  showFrame make_board this_fr windowx viewprt
    bundle
  errorHandler
    (make_composite_frame func channel notice_board
      menu make_board curr_fr
      (mergeVectorInNormalFrame created_frame
        labelx
      )
    )
  where
  this_fr = MkFrame labelx EmptyFrame
labelx = LabelToVector2 labell
labell = MkLabel strx (Pt2 10.0 50.0)
  windowx viewprt
)
)
mergeVectorInNormalFrame :: Frame -> Vector2 -> Frame
mergeVectorInNormalFrame f EmptyVector2 = f
mergeVectorInNormalFrame f v = MkFrame v f

--*****
distancex :: Point2 -> Point2 -> Float
distancex (Pt2 x1 y1) (Pt2 x2 y2) =
  sqrt (((x1 - x2)*(x1 - x2)) +
    (((y1 - y2)*(y1 - y2))))
-- RETURN A NEW FRAME BY DISPLACING BACK THE GIVEN FRAME BY
-- THE GIVEN POINT ::
displaceBack :: Frame -> Point2 -> Frame
displaceBack new_frame (Pt2 x y) =
  moveFrame new_frame (Pt2 xx yy)
where
    xx = (-1.0*x);
    yy = (-1.0*y)

---

button_to_command :: CommandDecode
button_to_command "Add" = 1
button_to_command "Delete" = 2
button_to_command "Help" = 3
button_to_command "Exit" = 4
button_to_command _ = 0

---

geom_button_to_command :: CommandDecode
geom_button_to_command "Scale" = 1
geom_button_to_command "Rotate" = 2
geom_button_to_command "Help" = 3
geom_button_to_command "Done" = 4
geom_button_to_command _ = 0

---

make_button_to_command :: CommandDecode
make_button_to_command "Point" = 1
make_button_to_command "Line" = 2
make_button_to_command "Circle" = 3
make_button_to_command "Text" = 4
make_button_to_command "Done" = 5
make_button_to_command "None" = 0

---

label_string :: Frame -> String
label_string (MkFrame
    (LabelToVector2 (MkLabel strx _ _)
    )
) = strx
label_string _ = "None"

---

get_selected_label :: Frame -> Int
get_selected_label EmptyFrame = 0

get_selected_label
    (MkFrame (LabelToVector2 (MkLabel "HOUSES" _ _))
    ) = 1
get_selected_label
    (MkFrame (LabelToVector2 (MkLabel "TREES" _ _))
    ) = 2
get_selected_label
    (MkFrame (LabelToVector2 (MkLabel "POLES" _ _))
    ) = 3
get_selected_label
    (MkFrame (LabelToVector2 (MkLabel "COMPOSITE" _ _))
    ) = 4
get_selected_label
    (MkFrame (LabelToVector2 (MkLabel "VEHICLES" _ _))
    ) = 5
get_selected_label (MkFrame _ f) = get_selected_label f
workstationName :: String
workstationName = "nms.hls.com:0"

---
ws_bundle, notice_bundle, menu_bundle :: BundledAttributes
ws_bundle = [(ValuatorNeeded 1)]
notice_bundle = [(ValuatorNeeded 0): (WsWidth 500): (WsHeight 500): (WsName "Notice Board"): []]
menu_bundle = [(ValuatorNeeded 0): (WsWidth 500): (WsHeight 500): (WsName "Menu"): []]

bundle :: GraphicsContext
bundle = [(ClearBeforeDump True)]

windowx :: Window2
windowx = (MkWindow2 (Pt2 0.0 0.0) (Pt2 100.0 100.0))

notice_window :: Window2
notice_window = (MkWindow2 (Pt2 0.0 0.0) (Pt2 100.0 10.0))

menu_window :: Window2
menu_window = (MkWindow2 (Pt2 0.0 0.0) (Pt2 100.0 100.0))

viewprt :: Window2
viewprt = (MkWindow2 (Pt2 0.0 0.0) (Pt2 1.0 1.0))

strx :: String
strx = "Om Namah Shivayah"

main_menu_frame :: Frame
main_menu_frame =
    (UnionFrame (moveFrame highrise (Pt2 30.0 60.0))
        (UnionFrame (moveFrame house1 (Pt2 30.0 30.0))
            (moveFrame house2 (Pt2 70.0 30.0))))

house_menu :: Frame
house_menu =
    (UnionFrame (moveFrame dome_house (Pt2 70.0 60.0))
        (UnionFrame (moveFrame highrise (Pt2 40.0 60.0))
            (UnionFrame (moveFrame house1 (Pt2 30.0 30.0))
                (moveFrame house2 (Pt2 70.0 30.0))))

tree_menu :: Frame
tree_menu =
    (UnionFrame (moveFrame tree3 (Pt2 40.0 60.0))
        (UnionFrame (moveFrame tree1 (Pt2 30.0 30.0))
            (moveFrame tree2 (Pt2 70.0 30.0))))

pole_menu :: Frame
pole_menu = (UnionFrame (moveFrame pole1 (Pt2 30.0 30.0))
            (moveFrame pole2 (Pt2 70.0 30.0)))

car_menu :: Frame
car_menu = (UnionFrame (moveFrame car1 (Pt2 30.0 30.0))
             (moveFrame car2 (Pt2 70.0 30.0)))

139
main_panel :: Frame
main_panel = listToUnion main_button_list

geometry_panel :: Frame
geometry_panel = listToUnion geometry_button_list

make_panel :: Frame
make_panel = listToUnion make_button_list

main_button_list :: [Vector2]
main_button_list = (add_b:delete_b:help_b:exit_b:[]);

geometry_button_list :: [Vector2]
geometry_button_list = (scale_b:rotate_b:ghelp_b:done_b:[]);

make_button_list :: [Vector2]
make_button_list = (point_b:line_b:circle_b:text_b:mdone_b:[]);

add_b, delete_b, help_b, exit_b :: Vector2
add_b = LabelToVector2 (MkLabel "Add" (Pt2 10.0 3.0) windowx viewprt)

delete_b = LabelToVector2 (MkLabel "Delete" (Pt2 30.0 3.0)
windowx viewprt)

help_b = LabelToVector2 (MkLabel "Help" (Pt2 50.0 3.0)
windowx viewprt)

exit_b = LabelToVector2 (MkLabel "Exit" (Pt2 70.0 3.0) windowx viewprt)

---
scale_b, rotate_b, ghelp_b, done_b :: Vector2
scale_b = LabelToVector2 (MkLabel "Scale" (Pt2 10.0 3.0) windowx viewprt)

rotate_b = LabelToVector2 (MkLabel "Rotate" (Pt2 30.0 3.0) windowx viewprt)

ghelp_b = LabelToVector2 (MkLabel "Help" (Pt2 50.0 3.0) windowx viewprt)

done_b = LabelToVector2 (MkLabel "Done" (Pt2 70.0 3.0) windowx viewprt)

---
point_b, line_b, circle_b, text_b, mdone_b :: Vector2
point_b = LabelToVector2 (MkLabel "Point" (Pt2 10.0 3.0)
windowx viewprt)

line_b = LabelToVector2 (MkLabel "Line" (Pt2 30.0 3.0)
windowx viewprt)
circle_b = LabelToVector2 (MkLabel "Circle" (Pt2 50.0 3.0)
windowx viewprt)
text_b = LabelToVector2 (MkLabel "Text" (Pt2 70.0 3.0)
windowx viewprt)
mdone_b = LabelToVector2 (MkLabel "Done" (Pt2 90.0 3.0)
--- Object menu ::
object_menu :: Frame
object_menu =
  (UnionFrame (moveFrame house_menu_icon (Pt2 10.0 10.0))
   (UnionFrame (moveFrame tree_menu_icon (Pt2 50.0 10.0))
    (UnionFrame (moveFrame pole_menu_icon (Pt2 10.0 40.0))
     (UnionFrame (moveFrame composite_icon (Pt2 50.0 40.0))
      (moveFrame car_icon (Pt2 10.0 70.0))
    )
   )
  )

---

--- Object Menu Icons ::
house_menu_icon :: Frame
house_menu_icon =
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 8.0 0.0) (Pt2 0.0 5.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 8.0 0.0) (Pt2 21.0 5.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 5.0) (Pt2 21.0 5.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 5.0 15.0) (Pt2 16.0 15.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 5.0 5.0) (Pt2 5.0 15.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 16.0 5.0) (Pt2 16.0 15.0)))
   (MkFrame (LabelToVector2 (MkLabel "HOUSES" (Pt2 6.0 18.0)
      menu_window viewport)) EmptyFrame))))

---

tree_menu_icon :: Frame
tree_menu_icon =
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 15.0)
      (Pt2 13.0 15.0)))
   (MkFrame (LabelToVector2 (MkLabel "TREES" (Pt2 3.0 18.0)
      menu_window viewport))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 5.0 5.0)
      (Pt2 5.0 15.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 8.0 5.0)
      (Pt2 8.0 15.0)))
   (MkFrame (Arc2ToVector2 (MkArc2 (Pt2 6.5 5.0)
      5.0 0.0 360.0))
     EmptyFrame))))

---
pole_menu_icon :: Frame
pole_menu_icon =
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 3.0)
      (Pt2 12.0 3.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 18.0)
      (Pt2 12.0 18.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 5.0 0.0) (Pt2 7.0 0.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 5.0 0.0)
      (Pt2 5.0 18.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 7.0 0.0) (Pt2 7.0 18.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 3.0) (Pt2 5.0 18.0)))
   (MkFrame (Line2ToVector2 (MkLine2 (Pt2 7.0 18.0) (Pt2 12.0 3.0)
   )))
   (MkFrame (LabelToVector2 (MkLabel "POLES" (Pt2 3.0 21.0)
     menu_window viewport))

141
---
car_icon :: Frame
car_icon =
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 2.0 0.0) (Pt2 7.0 0.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 5.0) (Pt2 22.0 5.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 10.0) (Pt2 22.0 10.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 5.0) (Pt2 0.0 10.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 2.0 0.0) (Pt2 2.0 10.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 7.0 0.0) (Pt2 7.0 5.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 22.0 5.0) (Pt2 22.0 10.0))
   ))
  (MkFrame (Arc2ToVector2 (MkArc2 (Pt2 5.0 10.0) 3.0 180.0 180.0)))
  (MkFrame (Arc2ToVector2 (MkArc2 (Pt2 15.0 10.0) 3.0 180.0 180.0))
   )
  (MkFrame (LabelToVector2 (MkLabel "VEHICLES" (Pt2 8.0 15.0)
      menu_window viewprt))
  
  EmptyFrame))))))))
---
composite_icon :: Frame
composite_icon =

  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 1.0 1.0) (Pt2 12.0 1.0)))
  )
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 12.0 1.0) (Pt2 12.0 12.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 12.0 12.0) (Pt2 1.0 12.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 1.0 12.0) (Pt2 12.0 12.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 12.0 12.0) (Pt2 12.0 12.0))
   ))
  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 12.0 12.0) (Pt2 12.0 12.0))
   ))
  (MkFrame (LabelToVector2 (MkLabel "COMPOSITE" (Pt2 1.0 15.0)
      menu_window viewprt))
  
  EmptyFrame))))))
--- House Structures ::
housel :: Frame
housel =

  (MkFrame (Line2ToVector2 (MkLine2 (Pt2 0.0 8.0) (Pt2 20.0 0.0)))
  )
  (MkFrame
   (Line2ToVector2 (MkLine2 (Pt2 0.0 8.0) (Pt2 20.0 8.0))
   ))
  (MkFrame
   (Line2ToVector2 (MkLine2 (Pt2 8.0 18.0) (Pt2 18.0 18.0))
   ))
  (MkFrame
   (Line2ToVector2 (MkLine2 (Pt2 11.0 12.0) (Pt2 15.0 12.0))
   ))
  (MkFrame
   (Line2ToVector2 (MkLine2 (Pt2 11.0 12.0) (Pt2 11.0 18.0))
   )

142
highrise :: Frame
highrise =
  ( MkFrame
    ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 0.0 ) ( Pt2 15.0 0.0 ) )
      ( MkFrame
        ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 4.0 ) ( Pt2 15.0 4.0 ) )
          ( MkFrame
            ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 8.0 ) ( Pt2 15.0 8.0 ) )
              ( MkFrame
                ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 12.0 ) ( Pt2 15.0 12.0 ) )
                  ( MkFrame
                    ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 16.0 ) ( Pt2 15.0 16.0 ) )
                      ( MkFrame
                        ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 20.0 ) ( Pt2 15.0 20.0 ) )
                          ( MkFrame
                            ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 24.0 ) ( Pt2 15.0 24.0 ) )
                              ( MkFrame
                                ( Line2ToVector2 ( MkLine2 ( Pt2 5.0 21.0 ) ( Pt2 10.0 21.0 ) )
                                  ( MkFrame
                                    ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 0.0 ) ( Pt2 0.0 24.0 ) )
                                      ( MkFrame
                                        ( Line2ToVector2 ( MkLine2 ( Pt2 5.0 0.0 ) ( Pt2 5.0 20.0 ) )
                                          ( MkFrame
                                            ( Line2ToVector2 ( MkLine2 ( Pt2 10.0 0.0 ) ( Pt2 10.0 20.0 ) )
                                              ( MkFrame
                                                ( Line2ToVector2 ( MkLine2 ( Pt2 15.0 0.0 ) ( Pt2 15.0 24.0 ) )
                                                  ( MkFrame
                                                    ( Line2ToVector2 ( MkLine2 ( Pt2 5.0 21.0 ) ( Pt2 5.0 24.0 ) )
                                                      ( MkFrame
                                                        ( Line2ToVector2 ( MkLine2 ( Pt2 7.0 21.0 ) ( Pt2 7.0 24.0 ) )
                                                          ( MkFrame
                                                            ( Line2ToVector2 ( MkLine2 ( Pt2 8.0 21.0 ) ( Pt2 8.0 24.0 ) )
                                                              ( MkFrame
                                                                ( Line2ToVector2 ( MkLine2 ( Pt2 10.0 21.0 ) ( Pt2 10.0 24.0 )
                                                                  ( EmptyFrame))))))))))))))))))))))))))
  )

dome_house :: Frame
dome_house =
  ( MkFrame
    ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 18.0 ) ( Pt2 20.0 18.0 ) )
      ( MkFrame
        ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 10.0 ) ( Pt2 20.0 10.0 ) )
          ( MkFrame
            ( Line2ToVector2 ( MkLine2 ( Pt2 0.0 10.0 ) ( Pt2 0.0 18.0 ) )
              ( MkFrame
                ( Line2ToVector2 ( MkLine2 ( Pt2 8.0 18.0 ) ( Pt2 8.0 10.0 ) )
                  ( MkFrame
                    ( Line2ToVector2 ( MkLine2 ( Pt2 10.0 18.0 ) ( Pt2 10.0 10.0 ) )
                      ( MkFrame
                        ( Line2ToVector2 ( MkLine2 ( Pt2 12.0 18.0 ) ( Pt2 12.0 10.0 ) )
                          ( MkFrame
                            ( Line2ToVector2 ( MkLine2 ( Pt2 20.0 18.0 ) ( Pt2 20.0 10.0 ) )
                              ( MkFrame
                                ( Arc2ToVector2 ( MkArc2 ( Pt2 10.0 10.0 ) 10.0 0.0 180.0 )
                                  ( EmptyFrame))))))))))))))))
  )
--- Trees :

tree1 :: Frame

tree1 =
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 0.0 15.0) (Pt2 13.0 15.0)))
  (MkFrame
    (Line2ToVector2 (MkLine2 (Pt2 5.0 5.0) (Pt2 5.0 15.0)))
    (MkFrame
      (Line2ToVector2 (MkLine2 (Pt2 8.0 5.0) (Pt2 8.0 15.0)))
      (Arc2ToVector2 (MkArc2 (Pt2 6.5 5.0) 5.0 0.0 360.0)
        EmptyFrame))))


tree2 :: Frame

tree2 =
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 0.0 15.0) (Pt2 12.0 15.0)))
  (MkFrame
    (Line2ToVector2 (MkLine2 (Pt2 5.0 5.0) (Pt2 5.0 15.0)))
    (MkFrame
      (Line2ToVector2 (MkLine2 (Pt2 7.0 5.0) (Pt2 7.0 15.0)))
      (Line2ToVector2 (MkLine2 (Pt2 3.0 7.0) (Pt2 9.0 7.0)))
      (MkFrame
        (Line2ToVector2 (MkLine2 (Pt2 6.0 0.0) (Pt2 3.0 7.0)))
        (MkFrame
          (Line2ToVector2 (MkLine2 (Pt2 6.0 0.0) (Pt2 9.0 7.0)))
          (MkFrame
            (Line2ToVector2 (MkLine2 (Pt2 3.0 9.0) (Pt2 9.0 9.0)))
            (MkFrame
              (Line2ToVector2 (MkLine2 (Pt2 6.0 2.0) (Pt2 3.0 9.0)))
              (MkFrame
                (Line2ToVector2 (MkLine2 (Pt2 6.0 2.0) (Pt2 9.0 9.0)))
                EmptyFrame)))))))))


tree3 :: Frame

tree3 =
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 0.0 12.0) (Pt2 12.0 12.0)))
  (MkFrame
    (Line2ToVector2 (MkLine2 (Pt2 5.0 5.0) (Pt2 5.0 12.0)))
    (MkFrame
      (Line2ToVector2 (MkLine2 (Pt2 7.0 5.0) (Pt2 7.0 12.0)))
      (Line2ToVector2 (MkLine2 (Pt2 5.0 5.0) (Pt2 0.0 5.0)))
      (MkFrame
        (Line2ToVector2 (MkLine2 (Pt2 5.0 5.0) (Pt2 5.0 0.0)))
        (MkFrame
          (Line2ToVector2 (MkLine2 (Pt2 7.0 5.0) (Pt2 7.0 0.0)))
          (MkFrame
            (Line2ToVector2 (MkLine2 (Pt2 7.0 5.0) (Pt2 12.0 5.0)))
            (Line2ToVector2 (MkLine2 (Pt2 5.0 5.0) (Pt2 2.0 2.0)))
            (MkFrame
              (Line2ToVector2 (MkLine2 (Pt2 7.0 5.0) (Pt2 10.0 2.0)))
              EmptyFrame))))))))

145
pole1 :: Frame
pole1 =
  (MkFrame
   (Line2ToVector2 (MkLine2 (Pt2 6.0 12.0) (Pt2 8.0 12.0)))
   (MkFrame
    (Line2ToVector2 (MkLine2 (Pt2 6.0 0.0) (Pt2 8.0 0.0)))
    (MkFrame
     (Line2ToVector2 (MkLine2 (Pt2 4.0 2.0) (Pt2 10.0 2.0)))
     (MkFrame
      (Line2ToVector2 (MkLine2 (Pt2 2.0 5.0) (Pt2 12.0 5.0)))
      (MkFrame
       (Line2ToVector2 (MkLine2 (Pt2 6.0 0.0) (Pt2 6.0 12.0)))
       (MkFrame
        (Line2ToVector2 (MkLine2 (Pt2 8.0 0.0) (Pt2 8.0 12.0)))
        (EmptyFrame)))))
pole2 :: Frame
pole2 =
  (MkFrame
   (Line2ToVector2 (MkLine2 (Pt2 3.0 0.0) (Pt2 5.0 0.0)))
   (MkFrame
    (Line2ToVector2 (MkLine2 (Pt2 0.0 2.0) (Pt2 8.0 2.0)))
    (MkFrame
     (Line2ToVector2 (MkLine2 (Pt2 2.0 6.0) (Pt2 6.0 6.0)))
     (MkFrame
      (Line2ToVector2 (MkLine2 (Pt2 0.0 2.0) (Pt2 2.0 6.0)))
      (MkFrame
       (Line2ToVector2 (MkLine2 (Pt2 8.0 2.0) (Pt2 6.0 6.0)))
       (MkFrame
        (Line2ToVector2 (MkLine2 (Pt2 3.0 16.0) (Pt2 5.0 16.0)))
        (MkFrame
         (Line2ToVector2 (MkLine2 (Pt2 3.0 0.0) (Pt2 3.0 16.0)))
         (MkFrame
          (Line2ToVector2 (MkLine2 (Pt2 5.0 0.0) (Pt2 5.0 16.0)))
          (EmptyFrame))))))))

-- Cars ::
car1, truck :: Frame
car1 =
  (MkFrame
   (Line2ToVector2 (MkLine2 (Pt2 5.0 0.0) (Pt2 15.0 0.0)))
   (MkFrame
    (Line2ToVector2 (MkLine2 (Pt2 0.0 5.0) (Pt2 20.0 5.0)))
    (MkFrame
     (Line2ToVector2 (MkLine2 (Pt2 0.0 9.0) (Pt2 20.0 9.0)))
     (MkFrame
      (Line2ToVector2 (MkLine2 (Pt2 0.0 5.0) (Pt2 0.0 9.0)))
      (MkFrame
       (Line2ToVector2 (MkLine2 (Pt2 20.0 5.0) (Pt2 20.0 9.0)))
       (MkFrame
        (Line2ToVector2 (MkLine2 (Pt2 20.0 0.0) (Pt2 20.0 9.0)))
        (MkFrame
         (Line2ToVector2 (MkLine2 (Pt2 3.0 0.0) (Pt2 3.0 5.0)))
         (MkFrame
          (Line2ToVector2 (MkLine2 (Pt2 15.0 0.0) (Pt2 17.0 5.0)))
          (EmptyFrame))))))
(MkFrame
  (Arc2ToVector2 (MkArc2 (Pt2 5.0 9.0) 2.0 180.0 180.0))
(MkFrame
  (Arc2ToVector2 (MkArc2 (Pt2 15.0 9.0) 2.0 180.0 180.0)
   EmptyFrame)))
)

truck =
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 2.0 0.0) (Pt2 7.0 0.0)))
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 0.0 5.0) (Pt2 22.0 5.0)))
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 0.0 10.0) (Pt2 22.0 10.0)))
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 0.0 5.0) (Pt2 0.0 10.0)))
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 2.0 0.0) (Pt2 2.0 10.0)))
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 7.0 0.0) (Pt2 7.0 5.0)))
(MkFrame
  (Line2ToVector2 (MkLine2 (Pt2 22.0 5.0) (Pt2 22.0 10.0)))
(MkFrame
  (Arc2ToVector2 (MkArc2 (Pt2 5.0 10.0) 3.0 180.0 180.0))
    (Arc2ToVector2 (MkArc2 (Pt2 15.0 10.0) 3.0 180.0 180.0)
     EmptyFrame)))))))

---

noticeFunction :: String -> Frame
noticeFunction strx =
  MkFrame (LabelToVector2
    (MkLabel strx (Pt2 5.0 5.0) notice_window viewprt))
  EmptyFrame

initial_mess,delete_mess :: String
initial_mess =
  "Welcome :: Select an operation by pressing the button"
delete_mess =
  " Click on the Picture Segment to be Deleted ...    "
add_mess =
  "Click on the Picture Segment to be selected ...    "
ext_mess =
  "    [ Call Exit :: Closing Connexion ]  "
delete_done_mess =
  "     [    Segment Deleted   ]    "
geom_mess =
  "  Click on the desired transformation or DONE when done"
main_help_mess =
  " Add: Add new Pic; Delete: Erase Pic; Ex it: Call Exit"
geomHelp_mess =
  " Scale: Scale Picture, Rotate: Rotate Pic, Done: Done."
geom_done_mess =
  " Transformed Image: Now click to place it on canvas "
make_done_mess =
  " Composite Frame Constructed :: Returning...    "
scale_pivot_mess =
  " Click on the Pivot for Scale Operation ...    "

147
rotate_pivot_mess =
   " Click on the Pivot for Rotate Operation ...  
place_mess =
   " Click to position the new Frame ....  
make_point_mess =
   " Click on the point to be included in the Frame...
make_line_mess =
   " Click on the first end of the line...
make_line_end_mess =
   " Click to mark the end of the line...
done_line_mess =
   " [ Line recorded into Frame ]
make_circle_mess =
   " Click on the centre of the circle...
make_radius_mess =
   " Click on the periphery of the circle...
done_circle_mess =
   " [ Circle recorded into Frame ]
del_err_mess =
   " Nothing to delete :: Canvas is empty !
no_del_mess =
   " No Frame Selected for Deletion !
make_err_mess =
   " No command selected :: Please retry !
nobutton_err_mess =
   " No Button Selected !

abnormalExit :: Dialogue
abnormalExit =
   appendChan stdout "[Abnormal Termination]" abort done

errorHandler :: IOError -> Dialogue
eroErrorHandler err_msg = fatalErrorHandler (get_string err_msg)

get_string :: IOError -> String
get_string (OtherError str) = str
get_string _ = "Std Error"

point2InWindow2 :: Window2 -> Point2 -> Bool
point2InWindow2 (MkWindow2 (Pt2 x1 y1) (Pt2 xh yh)) (Pt2 x y) =
   ((x1 <= x) && (x <= xh) && (y1 <= y) && (y <= yh))

-- Map a point from one window to another ::
viewportToWorld :: Window2 -> Window2 -> Point2 -> Point2
viewportToWorld (MkWindow2 (Pt2 x11 y11) (Pt2 xh1 yh1))
   (MkWindow2 (Pt2 x12 y12) (Pt2 xh2 yh2)) (Pt2 x y) =
   (Pt2 xx yy)

   where {
      xx = xaspect_ratio*(x - x11) + x11 ;
yy = yaspect_ratio*(y - y11) + y11 ;
xaspect_ratio = ((xh2 - x12)/(xh1 - x11));
yaspect_ratio = ((yh2 - y12)/(yh1 - y11))
   }

--************
viewportToWorld :: Point2 -> Window2 -> Window2 -> Point2
viewportToWorld point v w

148
| point2InWindow2 v point = viewportToWindow v w point |
| otherwise = EmptyPoint2 |

-------------
equalVec :: Vector2 -> Vector2 -> Bool
equalVec EmptyVector2 EmptyVector2 = True
equalVec (Pt2ToVector2 (Pt2 x1 y1)) (Pt2ToVector2 (Pt2 x2 y2)) = ((x1 == x2) && (y1 == y2))

equalVec (Line2ToVector2 (MkLine2 (Pt2 x1 y1) (Pt2 x2 y2)))
(Line2ToVector2 (MkLine2 (Pt2 xx1 yy1) (Pt2 xx2 yy2))) = 
((x1 == xx1) && (x2 == xx2) &&
 (y1 == yy1) && (y2 == yy2))

equalVec (Arc2ToVector2 (MkArc2 (Pt2 cx cy) r a l s))
(Arc2ToVector2 (MkArc2 (Pt2 cxx cyy) rr aal ss)) =
((cx == cxx) && (cy == cyy) && (r == rr) &&
 (a == aal) && (s == ss))

equalVec (LabelToVector2 (MkLabel str1 (Pt2 cx cy) _ _))
(LabelToVector2 (MkLabel str2 (Pt2 cxx cyy) _ _)) =
((cx == cxx) && (cy == cyy) && (str1 == str2))

equalVec _ _ = False
g-------------
equalFrame :: Frame -> Frame -> Bool
equalFrame EmptyFrame EmptyFrame = True
equalFrame (UnionFrame EmptyFrame f1) f2 = equalFrame f1 f2
equalFrame (MkFrame v1 f1) (MkFrame v2 f2) |
| equalVec v1 v2 = equalFrame f1 f2 |
| otherwise = False
equalFrame _ _ = False
g

filterFrame :: Frame -> Frame -> Frame
g
filterFrame EmptyFrame _ = EmptyFrame
g
filterFrame f EmptyFrame = f
g
filterFrame (MkFrame v1 f1) (MkFrame v2 f2) |
| equalFrame (MkFrame v1 f1) (MkFrame v2 f2) = EmptyFrame |
| otherwise = MkFrame v1 f1
g

filterFrame (UnionFrame f1 f2) f |
| equalFrame f1 f = f2 |
| otherwise = UnionFrame f1 (filterFrame f2 f)
g

nullFrame :: Frame -> Bool
nullFrame EmptyFrame = True
nullFrame (UnionFrame f1 f2) = ((nullFrame f1) && (nullFrame f2))
nullFrame _ = False
g

-- Find the least 'x' in a Frame ::

least_xy :: Int -> Frame -> Float -> Float

least_xy _ EmptyFrame x = x
least_x y n (MkFrame (Pt2ToVector2 (Pt2 x y)) f) s
  | n == 1 = if (s > x) then least_x y 1 f x
  else least_x y 1 f s
  | True   = if (s > y) then least_x y 2 f y
  else least_x y 2 f s

least_x y n (MkFrame (Line2ToVector2
  (MkLine2 (Pt2 x1 y1) (Pt2 x2 y2))) f) xs
  | n == 1 = if (xs > x) then
  least_x y 1 f x
  else
  least_x y 1 f xs
  where _
    x = min x1 x2
  | otherwise = if (xs > y) then
  least_x y 2 f y
  else
  least_x y 2 f xs
  where _
    y = min y1 y2

least_x y n (MkFrame (Arc2ToVector2
  (MkArc2 (Pt2 cx cy) _ _ _)) f) xs
  | n == 1 = if (xs > cx) then
  least_x y 1 f cx
  else
  least_x y 1 f xs
  | n == 2 = if (xs > cy) then
  least_x y 2 f cy
  else
  least_x y 2 f xs

least_x y n (MkFrame (LabelToVector2
  (MkLabel _ (Pt2 x y) _ _)) f) xs
  | n == 1 = if (xs > x) then
  least_x y 1 f x
  else
  least_x y 1 f xs
  | n == 2 = if (xs > y) then
  least_x y 2 f y
  else
  least_x y 2 f xs

least_x y n (UnionFrame f1 f2) xs =
  min (least_x y n f1 xs) (least_x y n f2 xs)

---

subtarctXComponent :: Frame -> Float -> Frame
subtarctXComponent EmptyFrame _ = EmptyFrame
subtarctXComponent (MkFrame v f) x =
  MkFrame (subtarctXfromVec v x) (subtarctXcomponent f x)
subtarctXComponent (UnionFrame f1 f2) x =
  UnionFrame (subtarctXcomponent f1 x)
  (subtarctXcomponent f2 x)
subtarctXfromVec :: Vector2 -> Float -> Vector2
subtarctXfromVec EmptyVector2 _ = EmptyVector2
subtarctXfromVec (Pt2ToVector2 (Pt2 x y)) xx =
  (Pt2ToVector2 (Pt2 (x - xx) y))

subtarctXfromVec (Line2ToVector2
  (MkLine2 (Pt2 x1 y1) (Pt2 x2 y2))) xx k
  (Line2ToVector2
    (MkLine2 (Pt2 (x1 - xx) y1) (Pt2 (x2 - xx) y2)))

subtarctXfromVec (Arc2ToVector2 (MkArc2 (Pt2 cx cy) r a l s)) xx =
  (Arc2ToVector2 (MkArc2 (Pt2 (cx - xx) cy) r a l s))

subtarctXfromVec (LabelToVector2
  (MkLabel strx (Pt2 x y) w v)) xx k
  (LabelToVector2 (MkLabel strx (Pt2 (x - xx) y) w v))

---

subtarctYcomponent :: Frame -> Float -> Frame
subtarctYcomponent EmptyFrame _ = EmptyFrame
subtarctYcomponent (MkFrame v f y) y' =
  UnionFrame (subtarctYcomponent f y') (subtarctYcomponent f y)

---

subtarctYfromVec :: Vector2 -> Float -> Vector2
subtarctYfromVec EmptyVector2 _ = EmptyVector2
subtarctYfromVec (Pt2ToVector2 (Pt2 x y)) yy =
  (Pt2ToVector2 (Pt2 x (y - yy)))
subtarctYfromVec (Line2ToVector2
  (MkLine2 (Pt2 x1 y1) (Pt2 x2 y2))) yy k
  (Line2ToVector2
    (MkLine2 (Pt2 x1 (y1 - yy)) (Pt2 x2 (y2 - yy))))
subtarctYfromVec (Arc2ToVector2 (MkArc2 (Pt2 cx cy) r a l s)) yy k
  (Arc2ToVector2 (MkArc2 (Pt2 cx (cy - yy)) r a l s))
subtarctYfromVec (LabelToVector2
  (MkLabel strx (Pt2 x y) w v)) yy k
  (LabelToVector2 (MkLabel strx (Pt2 x (y - yy)) w v))

---

fatalErrorHandler :: String -> Dialogue
fatalErrorHandler err_message =
  appendChan stdout err_message abort done

---

listToUnion :: [Vector2] -> Frame
listToUnion [] = EmptyFrame
listToUnion (x:xs) =
  UnionFrame (MkFrame x EmptyFrame) (listToUnion xs)
Vita

S Ramakrishnan was born in New Delhi, India, on August 23, 1962. He obtained B.S. in Mathematics from University of Bangalore in 1982. After working for a year with Bharat Heavy Electricals Ltd., he attended the Indian Institute of Technology, Bombay, where he obtained his Masters degree in Computer Science and Engineering in 1988. He worked for two years as Graphics R&D Engineer with WIPRO Information Technology Ltd., a leading micro- and minicomputer manufacturer of India.

He is currently employed as a Software Engineer with Hughes LAN Systems Inc., Mountain View, California.