# Programming with InterViews

Mark Linton
Silicon Graphics
2011 N. Shoreline Blvd.
P.O. Box 7311
Mountain View, CA 94039-7311
linton@sgi.com

## Outline

What is InterViews 3.0?

A simple application (hi mom!)

Menus et al.

Inside a glyph

A simple previewer

## Goals

- Make user interfaces easier to build
  - Tools should build applications faster
  - Need primitives and applications
- Build real applications
  - Document, drawing editors
- Make reusable components
  - Composition mechanisms
- Leverage technology
  - C++
  - Current/future hardware

**InterViews = Inter**active **Views**

Native C++

Glyphs: lightweight, shareable objects

Sophisticated layout objects

Resolution-independent graphics

Overlays

Incremental update

Graphical editing framework

## Availability

Unrestricted copyright (just like X)

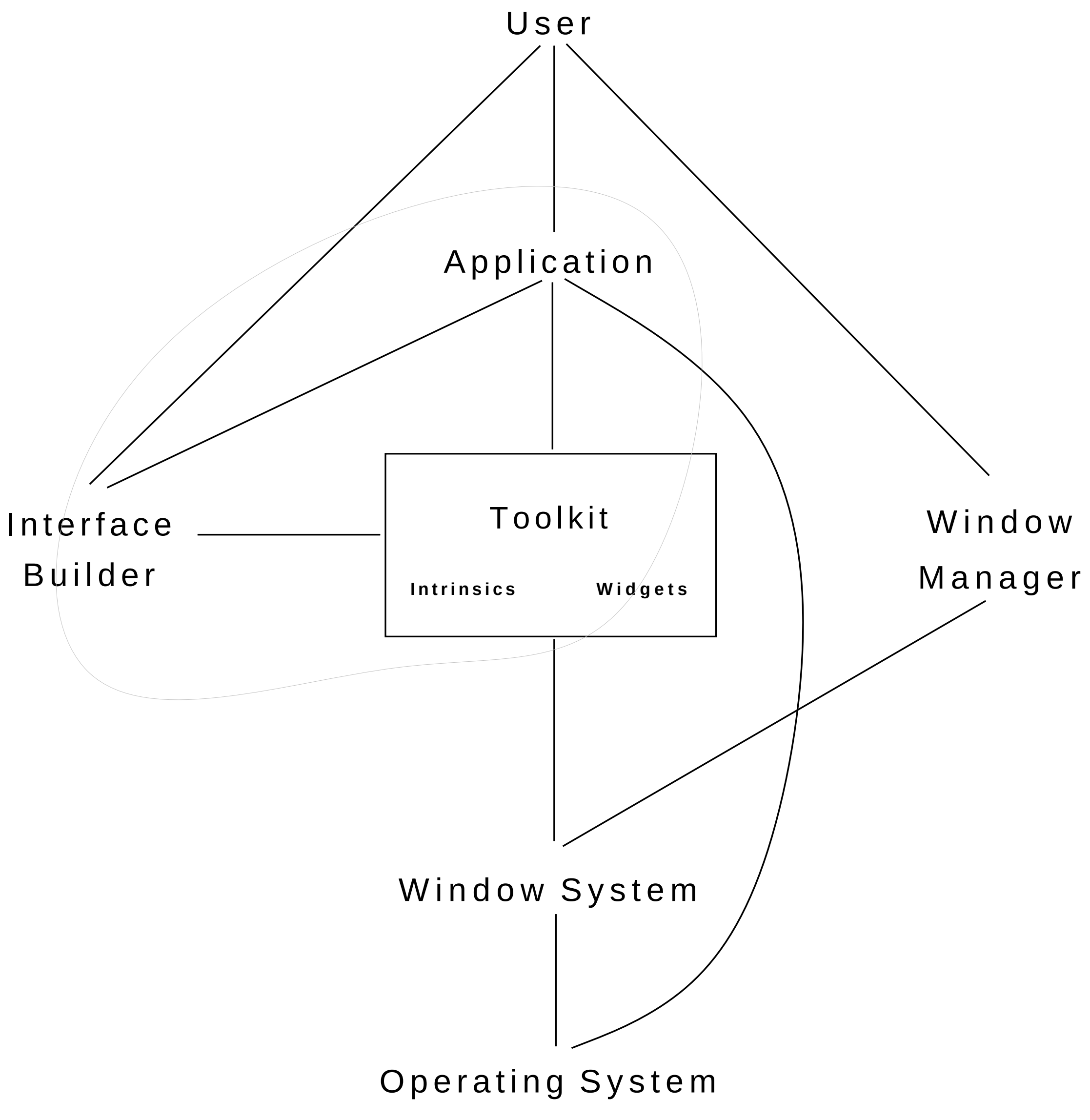
Anonymous ftp to sgi.com

graphics/interviews/3.0/beta.tar.Z

Requires >= C++ 2.0, X11R4

Runs on SGI, HP, Sun, DEC, IBM, and all X/Unix platforms

# User interface toolkits



## Differences from other toolkits

### C++

- Efficient, object-oriented language

### Composition mechanisms

- Boxes and glue
- Alignment
- TeX compositor

### Graphics

- Transformations (including fonts)
- Resolution-independence (printing)
- Direct color
- Structured graphics

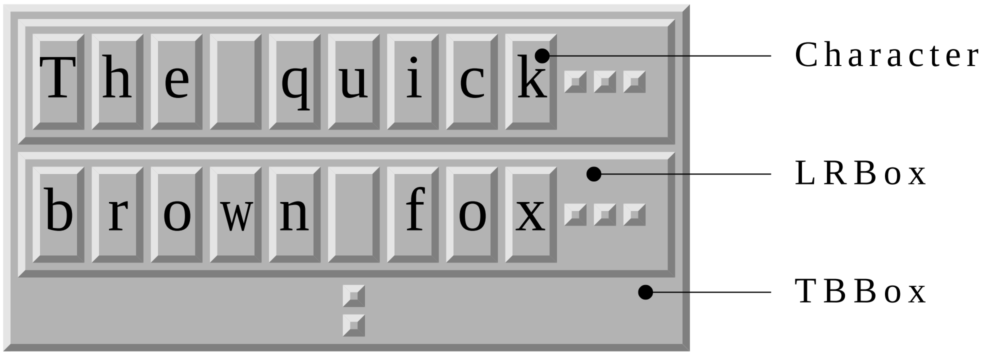
### Dynamic protocol

- Shareable objects
- Cheap objects

## User interface objects

- Toolkits tend to support control objects
  - Buttons, menus, scrollbars
- Users tend to manipulate data objects
  - Text, graphics, tables
- Data objects are much more plentiful
  - O(100,000) objects

## A text view ...




View structure reflects text structure

- Glyph per character
- Composite glyph per line

# TextView code

```
TextView::TextView(FILE* file) {
   TBBox* page = new TBBox;
   LRBox* line = new LRBox;
   int c;
   while ((c = getc(file)) != EOF) {
      if (c == ’\n’) {
         page->append(line);
         line = new LRBox;
      } else {
         line->append(new Character(c));
      }
   }
   body(page);
}
```

# ... with multiple fonts ...

```
*****

概要

ここで説明する各関数は Xlib の文字列のサイズの問い合わせ・
描画などを行う関数を EUC 文字列が扱えるように拡張したものです。
拡張された関数では環境変数 CSWIDTH を参照しますので、
関数を呼びだす前に CSWIDTH を正しく設定する必要があります。
また、プログラム中では次のファイルをインクルードして下さい。

    #include <X11/XJEuc.h>

関数の引数などのプログラム・
インタフェースはできる限り Xlib に合わせていますので、
特にことわり書きのないものについては Xlib の関数を参照して下さい。

プログラム・インタフェースで Xlib と大きく違う
点は XFontStruct 構造体の取り扱いです。
拡張された関数では XFontStruct 構造体の代わりに次の構造体を使用して下さい。

    typedef struct {
        XFontStruct     *font_std[4];   /* standard font structure table */
        XFontStruct     *font_ext[4];   /* extension font structure table */
        int             ascent;         /* maximum ascent */
        int             descent;        /* maximum descent */
    } XJeFontStruct;
```

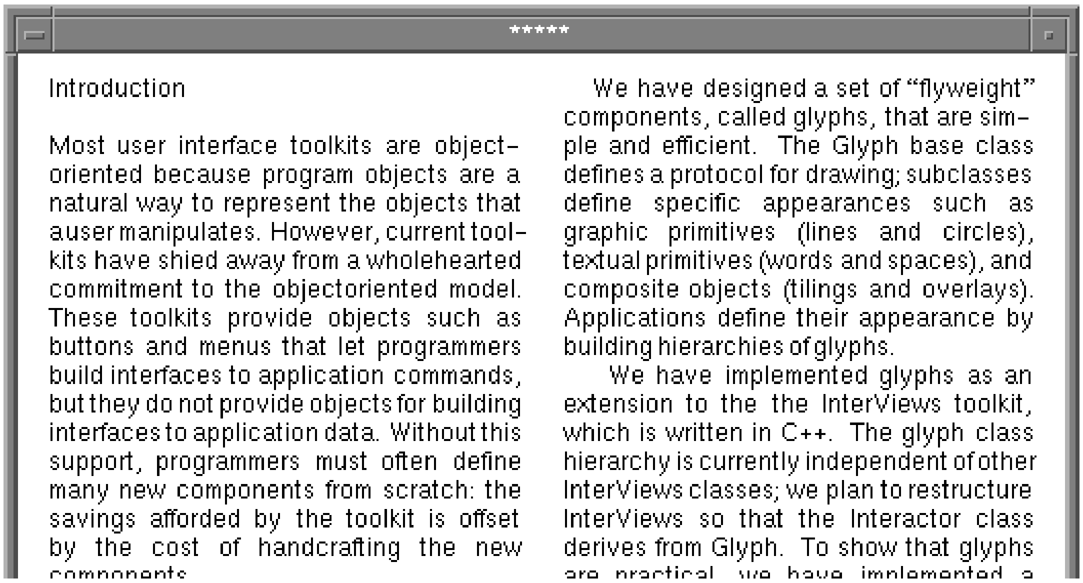
## Same simple structure but ...

- ASCII characters use ASCII-encoded font
- Kanjii characters use JIS-encoded font

# EUC TextView code

```
  Font* kf;
  Font* af;
  while ((c = getc(file)) != EOF) {
    if (c == '\n') {
      page->append(line);
      line = new LRBox;
+   } else if (!isascii(c)) {
+     line->append(
        new Character(tojis(c, getc(file)), kf)
      );
    } else {
      line->append(new Character(c, af));
    }
  }
```

# . . . and formatting



*****

Introduction

Most user interface toolkits are object-oriented because program objects are a natural way to represent the objects that a user manipulates. However, current toolkits have shied away from a wholehearted commitment to the objectoriented model. These toolkits provide objects such as buttons and menus that let programmers build interfaces to application commands, but they do not provide objects for building interfaces to application data. Without this support, programmers must often define many new components from scratch: the savings afforded by the toolkit is offset by the cost of handcrafting the new components.

We have designed a set of “flyweight” components, called glyphs, that are simple and efficient. The Glyph base class defines a protocol for drawing; subclasses define specific appearances such as graphic primitives (lines and circles), textual primitives (words and spaces), and composite objects (tilings and overlays). Applications define their appearance by building hierarchies of glyphs.

We have implemented glyphs as an extension to the the InterViews toolkit, which is written in C++. The glyph class hierarchy is currently independent of other InterViews classes; we plan to restructure InterViews so that the Interactor class derives from Glyph. To show that glyphs are practical, we have implemented a

## High-level help from a toolkit

- Line and column breaks
- Hyphenation

# Formatted TextView code

```
Composition* page = new LRComposition(
  new TBBox, new TeXCompositor, nil, width
);

while ((c = getc(file)) != EOF) {
  if (c == '\n') {
    page->append(
      new Discretionary(
        PenaltyGood,
        new HGlue(fil), new HGlue(0, fil), nil, nil
      )
    );
  } else if (c == ' ') {
    page->append(
      new Discretionary(
        0, new Character(' '), new HGlue(0, 0),
        nil, nil
      )
    );
  } else {
    page->append(new Character(c));
  }
}
```

## Glyphs are dynamic

Pass state instead of requiring storage

virtual void draw(Canvas*, const Allocation&);

Small objects

8 bytes overhead

Shareable objects

One character ‘a’ per document

Table entries in row and column layout
Menu items

## Glyphs are easy to use

- Characters –> LRComposition
  - Use TeXCompositor to determine breaks
- Lines –> TBComposition
  - Use TeXCompositor to determine breaks
- Columns –> LRComposition
  - Use TeXCompositor to determine breaks
- Pages
  - Use deck to flip between pages

## Glyph performance is good

Adequate response time
(10-mip workstation, framebuffer)

Doc uses ~3Mb data for 50 pages
(double column, single-spaced)

Dynamic protocol makes tuning easy
Too slow –> store information
Too big –> compute information

# Document editor

./intro.doc

File Edit Font Size Color Alignment Macro Item Special

## Glyphs: Flyweight Objects for User Interfaces

### 6 Introduction

Most user interface toolkits are object-oriented because program objects are a natural way to represent the objects that a user manipulates. However, current toolkits have shied away from a whole-hearted commitment to the object-oriented model. These toolkits provide objects such as buttons and menus that let program-mers build interfaces to application commands, but they do not provide objects for building interfaces to application data. Without this support, programmers must often define many new components from scratch; the savings afforded by the toolkit is offset by the cost of handcrafting the new components.

### 6.1 Overview

To offer the full benefits of an object-oriented model, a toolkit must encourage programmers to use objects for even the smallest components in the interface. Current toolkits let programmers build interfaces that contain hundreds of objects. But to reflect the fine-grained structure in application data, programmers may need hundreds of thousands of objects. For example, text is logically composed of characters, so a text view should be built from objects that represent individual characters. A moderate-sized document, such as a technical paper or a chapter in a book, will need at least 50,000 character objects.

### 6.2 Other toolkits

Most user interface toolkits are object-oriented because program objects are a natural way to represent the objects that a user manipulates. However, current toolkits have shied away from a wholehearted commitment to the object-oriented model. These toolkits provide objects such as buttons and menus that let program-mers build interfaces to application *commands*, but they do not provide objects for building interfaces to application *data*. Without this support, programmers must often define many new components from scratch; the savings afforded by the toolkit is offset by the cost of handcrafting the new components.

This approach can dramatically simplify an implementation, but it is practical only if objects are simple and cheap. The combination of faster hardware and more efficient object-oriented languages has now made it possible to use objects at this finer level of granularity without sacrificing interactive response.

We have designed a set of "flyweight" components, called glyphs, that are simple and efficient. The Glyph base class defines a protocol for drawing; subclasses define specific appearances such as graphic primitives (lines and



TextView object structure

```
TextView::TextView(FILE* file) {
  TBBox* page = new TBBox();
  LRBox* line = new LRBox();
  int c;
  while ((c = getc(file)) != EOF) {
    if (c == '\n') {
      page->append(line);
      line = new LRBox;
    } else {
      line->append(new Character(c));
    }
  }
  body(page);
}
```

A TextView built from glyphs

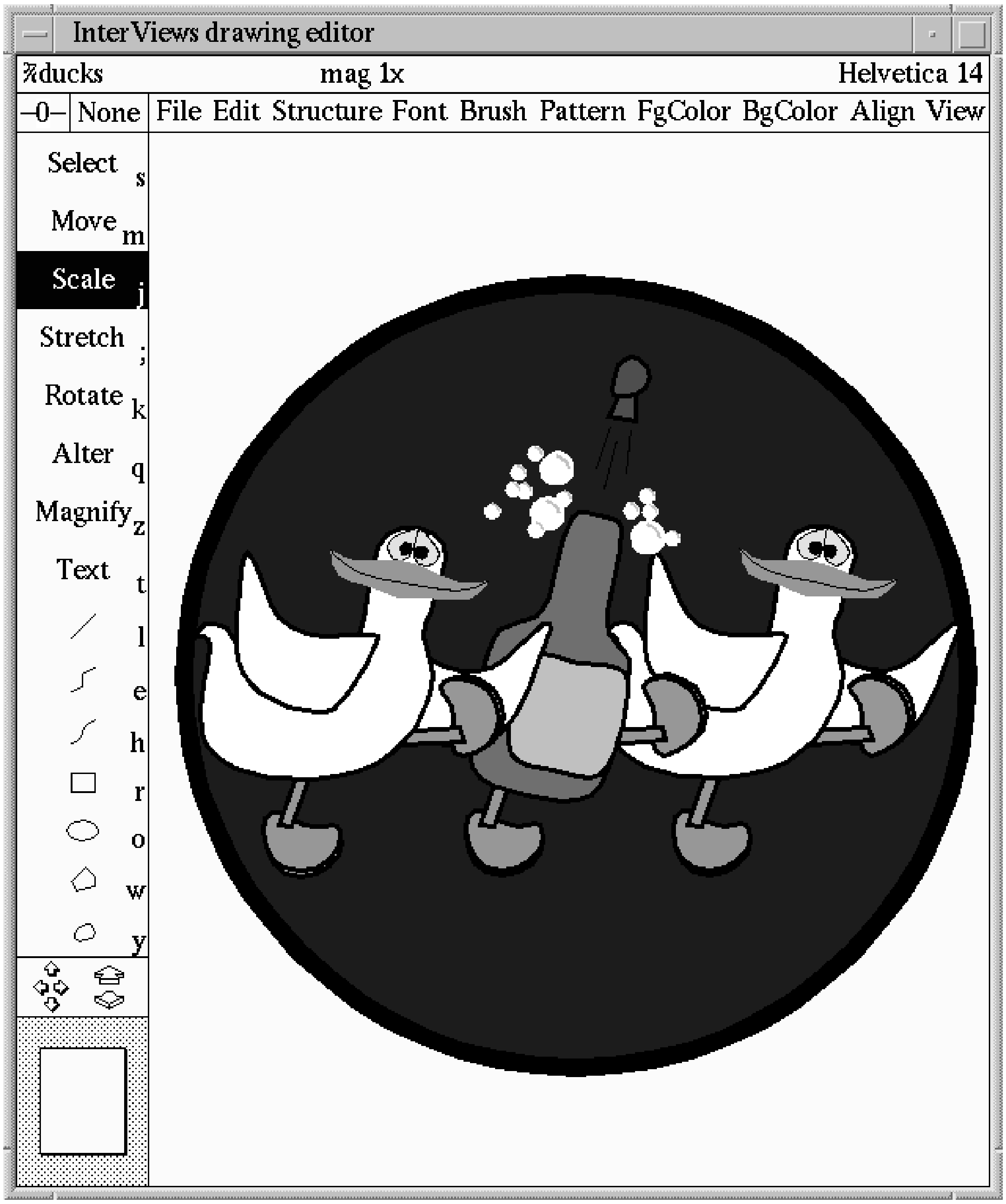
| | | |
|---|---|---|
| Request | **preference** | 10 |
| Allocates | size | 20 |
| Draw | appearance | 30 |

Glyph protocol

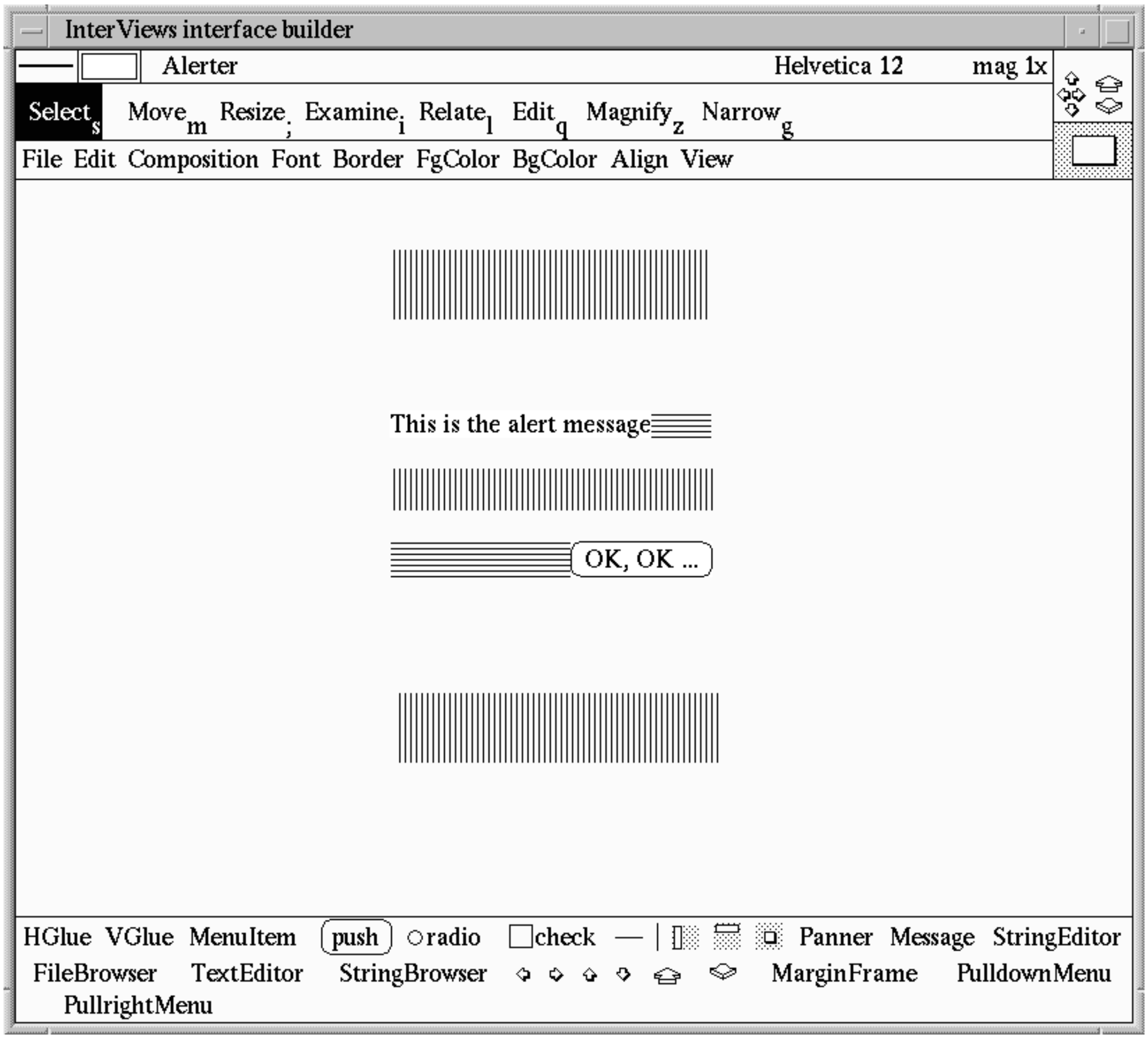
## Unidraw

- Components – objects in drawing
  - Connectors – graphical connectivity
  - State variables – dataflow
- Tools – prototypes/direct manipulation
- Commands – undo-able actions
- External representations – for domain
  - Postscript for all domains

# Idraw drawing editor



# Ibuild interface builder



## Current status

- Thousands of users world-wide
  - Companies, research labs, universities
- 3.0-beta available via anonymous ftp
- Working toward X Consortium standard
- Future work
  - Styles
  - Documents
  - Look+feel support
  - Multimedia
  - Multithreading

## A simple application

```
#include <InterViews/background.h>
#include <InterViews/label.h>
#include <InterViews/window.h>
#include <InterViews/world.h>

int main(int argc, char** argv) {
  World* world = new World("Himom", argc, argv);
  Window* window = new ApplicationWindow(
    new Background(
      new Label(
        "hi mom!", world->font(), world->foreground()
      ),
      world->background()
    )
  );
  window->map();
  world->run();
}
```

# Notation

```
typedef float Coord;
```

*Default units are printer's points*
*Relative to fonts (typically 75/72)*

```
typedef unsigned int boolean;
static const unsigned false = 0;
static const unsigned true = 1;

enum DimensionName {
  Dimension_X, Dimension_Y,
    ...
  Dimension_Undefined
};

#include <InterViews/enter-scope.h>
```

*Define name to* **iv***name*

No public or protected data

## Basics

Glyph – an object that draws

Canvas – a 2D place to draw

World – display/event manager

Window – managed canvas on display

Printer – canvas for printing

Requistion – desired size

Allocation – actual size

Layout – arrange components

Compositor – decide where to break

## Primitive glyphs

Character – code in font/color

Discretionary – changes on break

Glue, HGlue, VGlue – blank space

Image – displays a raster

Label – character string

Rule, HRule, VRule – visual separator

ShapeOf – identical size of another

Stencil – draw color through bitmap

## Composites

Box – arrange using layout

LRBox – tile in X dimension

TBBox – tile in Y dimension

Overlay – use superpose layout

Deck – only show “top” component

Page – position components explicitly

## MonoGlyph

Contains a single glyph (body)

MonoGlyph::append(g) calls
body()->append(g)

Background – draw color underneath
Border – draw color around edges
Center – align body in allocation
Composition – break using compositor
LRMarker – mark area of body
Margin – leave space around body
Patch – damage for update
Shadow – half-fill underneath
Target – pick sensitivity control
TransformSetter – concatenate matrix

## Graphics attributes

Bitmap – 2D boolean array

Brush – line thickness and style

Color – RGB + alpha

Font – family of character glyphs

Raster – 2D array of RGB + alpha

Transformer – transformation matrix

## Include directories

Intrinsics: <InterViews/*class*.h>

Look+feel: <IV-look/*class*.h>

X-dependent: <IV-X11/*class*.h>

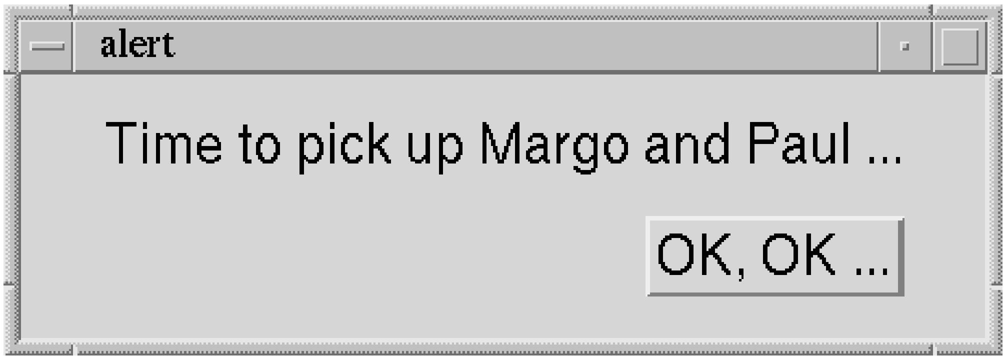
OS-dependent: <OS/*interface*.h>

Dispatching: <Dispatch/*class*.h>

# Imakefiles

```
#ifdef InObjectCodeDir

OBJS = main.o

Use_libInterViews()
ComplexProgramTarget(himom)

MakeObjectFromSrc(main)

#else

MakeInObjectCodeDir()

#endif
```

# Alert – display message in a dialog



## Making the message

```
Glyph* MakeMessage(Font* f, Color* fg) {
  char buffer[1024];
  Box* b = new TBBox;
  while (fgets(buffer, sizeof(buffer), stdin) != nil) {
    int n = strlen(buffer) - 1;
    if (buffer[n] == '\n') {
      buffer[n] = '\0';
    } else {
      ++n;
    }
    char* s = new char[n + 1];
    strcpy(s, buffer);
    b->append(
      new RMargin(new Label(s, f, fg), 0.0, fil, 0.0)
    );
  }
  return b;
}
```

## Boxing the message

```
Glyph* vspace = new VGlue(18.0, fil, 18.0);
Glyph* hspace = new HGlue(36.0, fil, 36.0);
  ...
new Background(
  new TBBox(
    new VCenter(vspace, 1.0),
    new LRBox(
      hspace,
      new Label(
        "hi mom!", w->font(), w->foreground()
      ),
      hspace
    ),
    vspace
  ),
  w->background()
)
```

## Making it customizable

```
static PropertyData props[] = {
   { "*quitbutton", "OK, OK ..." },
   { "*font", "9x15" },
   { "*transient", "on" },
   { nil }
};

static OptionDesc options[] = {
   { "font=", "*font", OptionValueAfter },
   { "button=", "*quitbutton", OptionValueAfter },
   { "-top", "*transient", OptionValueImplicit, "off" },
   { nil }
};

int main(int argc, char** argv) {
   World* w = new World(
      "Alert", argc, argv, options, props
   );
      ...
}
```

# Callbacks

```
class Action : virtual public Resource {
public:
   virtual void execute() = 0;
};

#define ActionCallback(T) ...

class ActionCallback(T) : public Action { \
public: \
   ActionCallback(T)(T*, ActionMemberFunction(T)*); \
   virtual void execute(); \
private: \
   T* obj_; \
   ActionMemberFunction(T)* func_; \
};

void ActionCallback(T)::execute() {
   (obj_->*func_)();
}
```

## Making the button

```
declare(ActionCallback,World)
implement(ActionCallback,World)

Glyph* label = new Margin(
   new Label(
      w->property_value("quitbutton"), f, fg
   ),
   3.0
);
   ...
new Button(
   new ActionCallback(World)(w, &World::quit),
   new ChoiceItem(
      new Bevel(label, new Outset(c)),
      new Bevel(label, new Inset(c))
   )
)
```

# Buttons and Telltales

```
class Button :
  public MonoGlyph, public PointerHandler {
public:
  Button(Action*, Telltale*);
};

class Telltale : public MonoGlyph {
public:
  Telltale(Glyph*);

  virtual void highlight(boolean);
  virtual void choose(boolean);
  virtual void enable(boolean);

  virtual boolean highlighted() const;
  virtual boolean chosen() const;
  virtual boolean enabled() const;
protected:
  virtual void redraw() const;
};
```

## ChoiceItem – Deck of looks

```
class ChoiceItem : public Telltale {
public:
  ChoiceItem(Glyph* normal, Glyph* pressed);
  ChoiceItem(Glyph*, Glyph*, Glyph* chosen);

  enum {
    is_enabled = 0x1, is_inside = 0x2,
    is_pressed = 0x4, is_chosen = 0x8
  };

  virtual void look(unsigned int bitset, Glyph*);
  virtual Glyph* look(unsigned int bitset) const;
};
```

# Bevels

```
typedef void (*Beveler)(
   Canvas*,
   const Color* light, const Color* medium,
   const Color* dark, Coord thickness,
   Coord left, Coord bottom, Coord right, Coord top
);

class Bevel : public MonoGlyph {
public:
   Bevel(Glyph*);
   Bevel(Glyph*, Beveler);
   Bevel(Glyph*, const BevelStyle*);
   Bevel(Glyph*, const BevelStyle*, Beveler);

   /* Predefined bevelers:
         rect,
         left_arrow, right_arrow, up_arrow, down_arrow
   */
};
```

## Bevel styles

```
class BevelStyle : virtual public Resource {
public:
  static Color* light(
    ColorIntensity r, ColorIntensity g, ColorIntensity b
  );
  static Color* dull(
    ColorIntensity r, ColorIntensity g, ColorIntensity b
  );
  static Color* dark(
    ColorIntensity r, ColorIntensity g, ColorIntensity b
  );

  virtual void thickness(Coord);
  virtual Coord thickness() const;

  virtual void set_colors(
    const Color* light,
    const Color* medium,
    const Color* dark
  );
  virtual const Color* light() const;
  virtual const Color* medium() const;
  virtual const Color* dark() const;
};
```

# Predefined bevel styles

```
class Inset : public BevelStyle {
public:
  Inset(const Color*);
};

class Outset : public BevelStyle {
public:
  Outset(const Color*);
};

class Flat : public BevelStyle {
public:
  Flat(const Color*);
};
```

## Putting it all together

```
int main(int argc, char** argv) {
  World* w = new World(
    "Alert", argc, argv, options, props
  );
  Font* f = w->font();
  Color* fg = w->foreground();
  ColorIntensity r, g, b;
  w->background()->intensities(w, r, g, b);
  Color* c = new Color(
    ColorIntensity(0.85*r),
    ColorIntensity(0.85*g),
    ColorIntensity(0.85*b)
  );
  Glyph* label = new Margin(
    new Label(w->property_value("quitbutton"), f, fg),
    3.0
  );
  Glyph* vspace = new VGlue(18.0, fil, 18.0);
  Glyph* hpsace = new HGlue(36.0, fil, 36.0);
```

```
Glyph* dialog = new Background(
  new TBBox(
    new VCenter(vspace, 1.0),
    new LRBox(
      hspace,
      new TBBox(
        new VCenter(MakeMessage(f, fg), 1.0),
        vspace,
        new LMargin(
          new Button(
            new ActionCallback(World)(
              w, &World::quit
            ),
            new ChoiceItem(
              new Bevel(label, new Outset(c)),
              new Bevel(label, new Inset(c))
            )
          ),
          0.0, fil, 0.0
        )
      ),
      hspace
    ),
    vspace
  ),
  c
);
```

```
boolean transient = w->property_is_on("transient");
Window* window;
if (transient) {
  TransientWindow* t = new TransientWindow(
    new Bevel(
      new Margin(dialog, 2.0),
      new Outset(c)
    )
  );
  t->transient_for(t);
  window = t;
} else {
  window = new ApplicationWindow(dialog);
}
window->place(w->width() / 2, w->height() / 2);
window->align(0.5, 0.5);
window->map();
w->RingBell(0);
w->run();
return 0;
```