Outline

What is InterViews 3.0?

A simple application (hi mom!)

Using menus and callbacks

Inside glyphs

Menu example

Pulldown menus

Callbacks

Damage checking

Sample execution

Menu class

```
class Menu : public MonoGlyph, public Handler {
public:
  Menu(Glyph*, float x1, float y1, float x2, float y2);

  virtual void item(
    GlyphIndex, Telltale*, Action*, Action*, Menu*,
    Window* = nil
  );
  virtual void add_item(
    Telltale*, Action*, Menu*, Window* = nil
  );

  virtual Telltale* telltale(GlyphIndex) const;
  virtual Action* action(GlyphIndex) const;
  virtual Menu* menu(GlyphIndex) const;
  virtual Window* window(GlyphIndex) const;

  void select(GlyphIndex);
};
```

Decorating menus

```
class DecoratingMenu : public Menu {
protected:
  DecoratingMenu(
    Glyph*, float x1, float y1, float x2, float y2
  );
public:
  virtual void set(
    GlyphIndex, Glyph*, Action*, Menu*, Window*
  );
  virtual void add(Glyph*, Action*, Menu*, Window*);

  virtual void separator(GlyphIndex, Glyph*);
  virtual void separator(GlyphIndex);
  virtual void add_separator(Glyph*);
  virtual void add_separator();
protected:
  virtual Telltale* make_item(Glyph*) = 0;
  virtual Glyph* make_separator();
};
```

Specific menus

```
class MenuBar : public DecoratingMenu {
public:
   MenuBar();
protected:
   virtual Telltale* make_item(Glyph*);
};

class Pulldown : public DecoratingMenu {
public:
   Pulldown();
protected:
   virtual Telltale* make_item(Glyph*);
};
```

Test application

```
class App {
public:
   App() { }
   int main(int argc, char** argv);

   void open();
   void save();
   void cut();
   void copy();
   void paste();
   void quit();
private:
   Glyph* MakeMenubar(Font*, Color*);
   Menu* MakePulldown(
      CommandInfo*, Font*, Color*
};

declare(ActionCallback,App);

int main(int argc, char** argv) {
   App a;
   return a.main(argc, argv);
}
```

Data-driven menus

```
struct CommandInfo {
   const char* str;
   ActionMemberFunction(App)* func;
};

CommandInfo filemenu[] = {
   { "Open", &App::open },
   { "Save", &App::save },
   { "", nil },
   { "Quit", &App::quit },
   { nil }
};

CommandInfo editmenu[] = {
   { "Cut", &App::cut },
   { "Copy", &App::copy },
   { "Paste", &App::paste },
   { nil }
};
```

Making the menubar

```
Glyph* App::MakeMenubar(Font* f, Color* fg) {
    MenuBar* m = new MenuBar;
    m->add(
        new Label("File", f, fg),
        MakePulldown(filemenu, f, fg)
    );
    m->add(
        new Label("Edit", f, fg),
        MakePulldown(editmenu, f, fg)
    );
    /* disable save*/
    m->menu(0)->telltale(1)->enable(false);
    return m;
}
```

Making a pulldown

```
Menu* App::MakePulldown(
   CommandInfo* info, Font* f, Color* fg
) {
   Pulldown* m = new Pulldown;
   for (CommandInfo* i = info; i->str != nil; i++) {
      if (i->str[0] == '\0') {
         m->add_separator();
      } else {
         m->add(
            new RMargin(
               new Label(i->str, f, fg),
               0.0, fil, 0.0
            ),
            new ActionCallback(App)(this, i->func)
         );
      }
   }
   return m;
}
```

Action routines

```
void App::open() {
   printf("open\n");
}

void App::save() {
   printf("save\n");
}
...
void App::quit() {
   printf("quit\n");
   World::current()->quit();
}
```

Incremental updates

Canvas collects damage area

Update traverses from root

Short-circuit looking at damage

Tradeoffs
- Cost of redraw
- Cost of damage check (time)
- Cost of bounding box (space)

Leaves typically don’t check

Damage checker

```
class DamageChecker : public MonoGlyph {
public:
  DamageChecker(Glyph*);

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

DamageChecker::DamageChecker(Glyph* g) :
  MonoGlyph(g) { }

void DamageChecker::draw(
  Canvas* c, const Allocation& a
) const {
  if (c->damaged(
    a.left(), a.bottom(), a.right(), a.top()
  ) {
    MonoGlyph::draw(c, a);
  }
}
```

Putting it all together

```
int App::main(int argc, char** argv) {
  World world("MenuTest", argc, argv, options, props);
  Color* fg = world.foreground();
  Color* bg = world.background();
  Font* f = world.font();
  Window* w = new ApplicationWindow(
    new TBBox(
      new VCenter(MakeMenubar(f, fg), 1.0),
      new Background(
        new TBBox(
          new DamageChecker(
            new Label("this is some text", f, fg)
          ),
          new DamageChecker(
            new Label("more text!", f, fg)
          )
        ),
        bg
      )
    )
  );
  w->map();
  world.run();
  delete w;
  return 0;
}
```

Inside a glyph

Glyph base class

Rule

Background

Bevel::rect

Button

Glyph base class

```
class Glyph : virtual public Resource {
public:
  virtual void request(Requisition&) const;
  virtual void allocate(
    Canvas*, const Allocation&, Extension&
  );
  virtual void draw(Canvas*, const Allocation&) const;
  virtual void print(Printer*, const Allocation&) const;
  virtual boolean pick(
    Canvas*, const Allocation&, Coord x, Coord y,
    Hit&, int depth
  );
  virtual Glyph* compose(GlyphBreakType);
```

Glyph components

```
virtual void append(Glyph*);
virtual void prepend(Glyph*);
virtual void insert(GlyphIndex, Glyph*);
virtual void remove(GlyphIndex);
virtual void replace(GlyphIndex, Glyph*);
virtual void change(GlyphIndex);

virtual GlyphIndex count() const;
virtual Glyph* component(GlyphIndex) const;
virtual void allotment(
   GlyphIndex index, DimensionName, Allotment&
) const;
```

Rule – visible separator

```
Rule::Rule(
   DimensionName d, const Color* c, Coord t
) : Glyph() {
   dimension_ = d;
   color_ = c;
   Resource::ref(color_);
   thickness_ = t;
}

Rule::~Rule() {
   Resource::unref(color_);
}

HRule::HRule(const Color* c, Coord t) :
   Rule(Dimension_Y, c, t) { }

VRule::VRule(const Color* c, Coord t) :
   Rule(Dimension_X, c, t) { }
```

Rule implementation

```
void Rule::request(Requsition& req) const {
   Requirement r(thickness_, 0, 0, 0);
   req.require(dimension_, r);
}

void Rule::allocate(
   Canvas*, const Allocation& a, Extension& ext
) {
   ext.xy_extents(a);
}

void Rule::draw(
   Canvas* c, const Allocation& a
) const {
   c->fill_rect(
      a.left(), a.bottom(), a.right(), a.top(), color_
   );
}
```

Canvas drawing interface

```
class Canvas {
public:
  virtual void new_path();
  virtual void move_to(Coord x, Coord y);
  virtual void line_to(Coord x, Coord y);
  virtual void curve_to(
    Coord x, Coord y,
    Coord x1, Coord y1, Coord x2, Coord y2
  );
  virtual void close_path();

  virtual void stroke(const Color*, const Brush*);
  virtual void fill(const Color*);

  virtual void character(
    const Font*, long ch, Coord width, const Color*,
    Coord x, Coord y
  );
  virtual void stencil(
    const Bitmap*, const Color*, Coord x, Coord y
  );
  virtual void image(const Raster*, Coord x, Coord y);
```

Clipping and transformations

```
virtual void push_transform();
virtual void transform(const Transformer&);
virtual void pop_transform();

virtual void push_clipping();
virtual void clip();
virtual void pop_clipping();
```

Convenience routines

```
virtual void line(
   Coord x1, Coord y1, Coord x2, Coord y2,
   const Color*, const Brush*
);
virtual void rect(
   Coord left, Coord bottom, Coord right, Coord top,
   const Color*, const Brush*
);
virtual void fill_rect(
   Coord left, Coord bottom, Coord right, Coord top,
   const Color*
);
virtual void clip_rect(
   Coord left, Coord bottom, Coord right, Coord top
);
```

Damage control

```
virtual void damage(
   Coord left, Coord bottom, Coord right, Coord top
);

virtual boolean damaged(
   Coord left, Coord bottom, Coord right, Coord top
) const;

virtual void damage_all();
virtual boolean any_damage() const;

virtual void redraw(
   Coord left, Coord bottom, Coord right, Coord top
);
virtual void repair();
```

Background – opaque underneath

```
Background::Background(
   Glyph* body, const Color* c
) : MonoGlyph(body) {
   color_ = c;
   Resource::ref(color_);
}

Background::~Background() {
   Resource::unref(color_);
}
```

Background implementation

```
void Background::allocate(
   Canvas* c, const Allocation& a, Extension& ext
) {
   MonoGlyph::allocate(c, a, ext);
   ext.extend(a);
}

void Background::draw(
   Canvas* c, const Allocation& a
) const {
   Coord left = a.left(), bottom = a.bottom();
   Coord right = a.right(), top = a.top();
   if (c->damaged(left, bottom, right, top)) {
      c->fill_rect(left, bottom, right, top, color_);
   }
   MonoGlyph::draw(c, a);
}

void Background::print(
   Printer* p, const Allocatino& a
) const {
   Coord left = a.left(), bottom = a.bottom();
   Coord right = a.right(), top = a.top();
   if (p->damaged(left, bottom, right, top)) {
      p->fill_rect(left, bottom, right, top, color_);
   }
   MonoGlyph::print(p, a);
}
```

Drawing on canvases

```
void Bevel::rect(
   Canvas* c, const Color* light, const Color* medium,
   const Color* dark, Coord thickness,
   Coord left, Coord bottom, Coord right, Coord top
) {
   /* background */
   c->fill_rect(left, bottom, right, top, medium);

   Coord left_inside = left + thickness;
   Coord bottom_inside = bottom + thickness;
   Coord right_inside = right - thickness;
   Coord top_inside = top - thickness;

   /* left edge */
   c->new_path();
   c->move_to(left, bottom);
   c->line_to(left, top);
   c->line_to(left_inside, top);
   c->line_to(left_inside, bottom);
   c->close_path();
   c->fill(light);
```

```
/* top edge */
c->new_path();
c->move_to(left_inside, top_inside);
c->line_to(left_inside, top);
c->line_to(right, top);
c->line_to(right, top_inside);
c->close_path();
c->fill(light);

/* right edge */
c->new_path();
c->move_to(right_inside, bottom_inside);
c->line_to(right_inside, top_inside);
c->line_to(right, top);
c->line_to(right, bottom);
/* bottom edge as part of same path */
c->line_to(left, bottom);
c->line_to(left_inside, bottom_inside);
c->line_to(right_inside, bottom_inside);
c->close_path();
c->fill(dark);
```

Adding input behavior

Dispatcher::dispatch

World::read

Window::receive

Window::target

Glyph::pick, Listener::pick

Hit::listen

Handler::event

PointerHandler

```
class PointerHandler : public Handler {
protected:
   PointerHandler();
public:
   virtual void sense(Event&);
   virtual void press(Event&);
   virtual void drag(Event&);
   virtual void release(Event&);
   virtual void commit(Event&);
   virtual void abort(Event&);
};
```

Events

world() – returns target display
handler() – returns appropriate handler
grab(Handler*) – snatch future events
ungrab(Handler*) – release

type() – motion, down, up, key, other
pointer_x(), pointer_y()
pointer_button()
mapkey(char*, unsigned int len) –
  map key event to string

Buttons

```
Button::Button(Action* a, Telltale* t) :
    MonoGlyph(nil), PointerHandler() {
    action_ = a;
    Resource::ref(action_);
    telltale_ = t;
    Resource::ref(telltale_);
    listener_ = new Listener(telltale_, this);
    listener_->sensor()->button(true, Event::any);
    listener_->sensor()->motion(true);
    body(listener_);
}

Button::~Button() {
    Resource::unref(action_);
    action_ = nil;
    Resource::unref(telltale_);
    telltale_ = nil;
}
```

```
void Button::drag(Event& e) {
   if (telltale_ != nil) {
      telltale_->highlight(
         listener_->picks(e.pointer_x(), e.pointer_y())
      );
   }
}

void Button::release(Event& e) {
   if (telltale_ != nil) {
      telltale_->highlight(false);
   }
   PointerHandler::release(e);
}

void Button::commit(Event& e) {
   if (action_ != nil &&
      listener_->picks(e.pointer_x(), e.pointer_y())
   ) {
      action_->execute();
   }
}
```

Separation of input and output

Button only defines input behavior

Telltale selects output behavior

Decks often implement switching

MonoGlyphs augment behavior
but are not visible structurally
(insert, remove, change, component)

Forcing an update

```
void Telltale::allocate(
   Canvas* c, const Allocation& a, Extension& ext
) {
   MonoGlyph::allocate(c, a, ext);
   ext.extend(a);
   canvas_ = c;
   extension_ = ext;
}

void Telltale::redraw() const {
   if (canvas_ != nil) {
      canvas_->damage(
         extension_.left(), extension_.bottom(),
         extension_.right(), extension_.top()
      );
   }
}
```

Summary

- Brief overview of InterViews 3.0
- Evolve hi-mom –> alert
- Use menus
- Defining glyphs
  - request, allocate, draw, print, pick
  - insert, remove, change, component
- MonoGlyphs to extend behavior
  - Input events

Where to go from here

Get 3.0-beta distribution

Build it (need C++ compiler, X)

Run doc

Look at papers, documentation

Write a program

Investigate Unidraw, ibuild

Read comp.windows.interviews

Plan for future

Styles

Xdefaults, document styles, structured graphics

Documents

Capabilities of doc in library

Multimedia

Composition of time-based objects

X Consortium working group

More commercial products

Power from simple ideas

- Native C++
- Glyphs
  - Lightweight, shareable
  - Separate input and output behavior
- Sophisticated layout
  - Boxes and glue
  - TeXCompositor
  - Overlays
- Resolution-independence
- Incremental update
- Graphical editing framework