TinIV — This is not InterViews
Strukturierte Graphik für **InterViews 3.1**

Clemens Wagner[1]
Universität/GHS-Siegen
FB 12 – Technische Informatik

Version vom 16. September 1993

[1]eMail: `clemens@server.informatik.uni-siegen.de`

Zusammenfassung

Die nachfolgend beschriebenen Klassen sind eine Erweiterung von **InterViews 3.1** für strukturierte Graphik. Sie sollen sowohl die Darstellung als auch die Verwaltung (z.B.: Erzeugen, Selektieren, Löschen u.v.m.) von einfachen und komplexeren graphischen Objekten ermöglichen. Sie sollen — ähnlich wie `Unidraw` unter **InterViews 2.6** — das Erstellen von graphischen Editoren unterstützen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die nachfolgend beschriebene Bibliothek für strukturierte Graphik unter **InterViews 3.1** spaltet sich im wesentlichen in drei Hierarchien auf: in Klassen für Graphik (siehe 2), Werkzeuge (siehe 3.1) und Kommandos (siehe 3.2).

Außerhalb dieser Hierarchien gibt es noch Klassen, die die Einbindung in **InterViews** erleichtern und Hilfsklassen.

1.1 Geschichte

Unter **InterViews 2.6** gibt es eine sehr mächtige Bibliothek für Graphic namens `Unidraw`, die man aber leider unter **InterViews 3.1** nur sehr schlecht verwenden kann. In der **InterViews 3.1** -Distribution gibt es allerdings ein Beispielprogramm, daß strukturierte Graphik unter **InterViews 3.1** hat. Es ist das Programm `morpher`.

TinIV ist aus diesem Beispiel mit sehr vielen Änderungen und Erweiterungen an den `morpher`–Quelltexten entstanden. So wurde *keine* Klasse ohne Veränderungen übernommen und einige Klassen und Klassenhierarchien neu hinzugefügt.

1.2 Notationen

Die Schreibweise der Klassendeklarationen lehnt sich an die Schreibweise in [iv3.1] an. Die Klassendeklaration wird dabei so abgebildet, wie sie in der C++-Headerdatei erscheint. Mit den Änderungen:

- Es werden nur die öffentlichen Methoden, Basisklassen und Instanzen der Klasse angegeben.
- Der Destruktor wird nicht mit angegeben.
- Statt des Schlüsselwortes `class` wird `interface` verwendet.
- Die Schlüsselworte `public`, `protected` und `private` entfallen.
- Virtuelle Methoden, die in einer Basisklasse deklariert wurden, werden in der abgeleiteten Klasse nicht aufgeführt.

Bei (abstrakten) Basisklassen werden unter Umständen geschützte Methoden der Klasse im Text erklärt.

Alle Klassen- und Methodennamen werden im Schriftstil `Typewriter` gesetzt. Ist jedoch der Oberbegriff für eine Klassenhierarchie gemeint, d.h. für die Basisklasse und alle ihre Ableitungen, so wird das deutsche Wort oder ein eingedeutschtes Wort verwendet. Beispiele:

- *Die Klasse* `Graphic` *enthält eine Methode* `fill`*, mit der man die Füllfarbe setzen kann.*
  Hier ist die Klasse `Graphic` gemeint.
- *In eine Polygraphik können (beliebig) viele andere Graphiken eingefügt werden.*
  Mit „Polygraphik“ sind sowohl Objekte vom Typ `PolyGraphic`, als auch deren Ableitungen gemeint. Analoges gilt für „Graphik“.

1.3 Attribute

Ein *Attribut* (siehe Abbildung 1.1) bestimmt das Aussehen einer Graphik. Sie können sowohl bei der Konstruktion als auch nachträglich gesetzt oder gelöscht werden.

Attribut	*Typ*	*Auswirkung*	*Bemerkung*
`transformer`	`Transformer`	Verformung	
`brush`	`Brush`	Liniendicke und -muster	nicht bei Textgraphiken
`stroke`	`Color`	Linienfarbe	nicht bei Textgraphiken
`fill`	`Color`	Füllfarbe	Farbe bei Textgraphiken
`font`	`Font`	Zeichensatz	nur bei Textgraphiken
`closed`	`boolean`	geschlossener Linienzug	nur bei Liniengraphiken
`curved`	`boolean`	gebogener Linienzug	nur bei Liniengraphiken

Abbildung 1.1: Mögliche Attribute einer Graphik

Eine Textgraphik ist eine Graphik vom Typ `Text` (siehe 2.1.6) und eine Liniengraphik ist eine Graphik, die aus einem Linienzug (z.B.: `Rectangle`, `Circle` usw.) besteht.

Mit dem `Transformer` kann die Graphik gedreht, gestreckt und/oder verschoben werden. Diese Operationen werden in Zukunft kurz *Verformung* bzw. *Deformation* genannt.

Das Attribut `brush` bestimmt die Liniendicke und ggf. das Muster des Randes einer Graphik. Die Farbe des Randes wird durch `stroke` festgelegt. Das Attribut `fill` steuert, ob die Graphik ausgefüllt werden soll. Bei Textgraphiken kann über `font` die Schriftart bestimmt werden.

Abbildung 1.2: Die Auswirkung des Attributes `curved`

In Bild 1.2 ist die Auswirkung des Attributes `curved` auf eine Liniengraphik dargestellt. Im linken Bild ist es auf `false` gesetzt, weswegen die Punkte (x0,y0) bis (x3,y3) miteinander durch Geraden verbunden werden. Ist das Attribut auf `true` gesetzt, so werden die Punkte als Stützpunkte einer Bézierkurve betrachtet (rechtes Bild).

Abbildung 1.3: Die Auswirkung des Attributes `closed`

Die Auswirkung des Attributes `closed` ist in Bild 1.3 dargestellt. Wenn es gesetzt ist, werden der erste und der letzte Punkt der Liniengraphik miteinander verbunden. Solche Liniengraphiken werden auch *geschlossen* genannt. Dementsprechend werden Graphiken, bei denen dieses

Attribut nicht gesetzt ist, *offen* genannt[1].

Eine Besonderheit stellen Polygraphiken (siehe 2.1.5) dar, die die Attribute `brush`, `stroke`, `fill` und `font` an die Graphiken, die sie enthalten, vererben, d.h.: Wenn eines dieser Attribut in einer Polygraphik gesetzt ist, dann ignoriert das Kind sein eigenes Attribut und beachtet nur das der Polygraphik. Auf Objekte vom Typ `PolyGraphic` haben die Attribute keine Auswirkung, weil sie nicht *selber* auf den Bildschirm zeichnen.

1.4 Benutzungsrechte

TinIV darf beliebig kopiert, verbreitet, genutzt und verändert werden. Es gelten allerdings folgende Einschränkungen:

- Es darf immer nur das vollständige Archiv weitergegeben werden.
- Wird veränderter Code weitergegeben, dann ist in diesem ein Hinweis auf **TinIV** und dessen Benutzungsrechte anzugeben. Der Kommentar zu Anfang jeder `*.cxx`– und `*.h`–Datei darf nicht gelöscht oder verändert werden.
- Es gelten alle Bedingungen, wie für die **InterViews 3.1** –Quelltexte, sofern sie nicht einem oben genannten Punkt widersprechen.

1.5 Systemvoraussetzungen und Haftungsausschluß

Die Klassenbibliothek setzt mindestens **InterViews 3.1** , UNIX System V mit X11R4 oder X11R5 sowie einen C++–Compiler (`AT&T 2.1` oder neuer) voraus. Sie wurde auf DECstations 3100 und 5000 unter Ultrix 4.2 und 4.3 getestet; eine Portierung auf andere Rechner oder UNIX-Versionen sollte aber (problemlos) möglich sein.

Ich übernehme keinerlei Haftung für Schäden, die durch diese Bibliothek verursacht wurden. Die Benutzung erfolgt also auf eigene Gefahr.

[1] Offene Liniengraphiken, bei denen das Füllattribut gesetzt ist, werden wie geschlossene Liniengraphiken ausgefüllt.

Kapitel 2

Graphikklassen

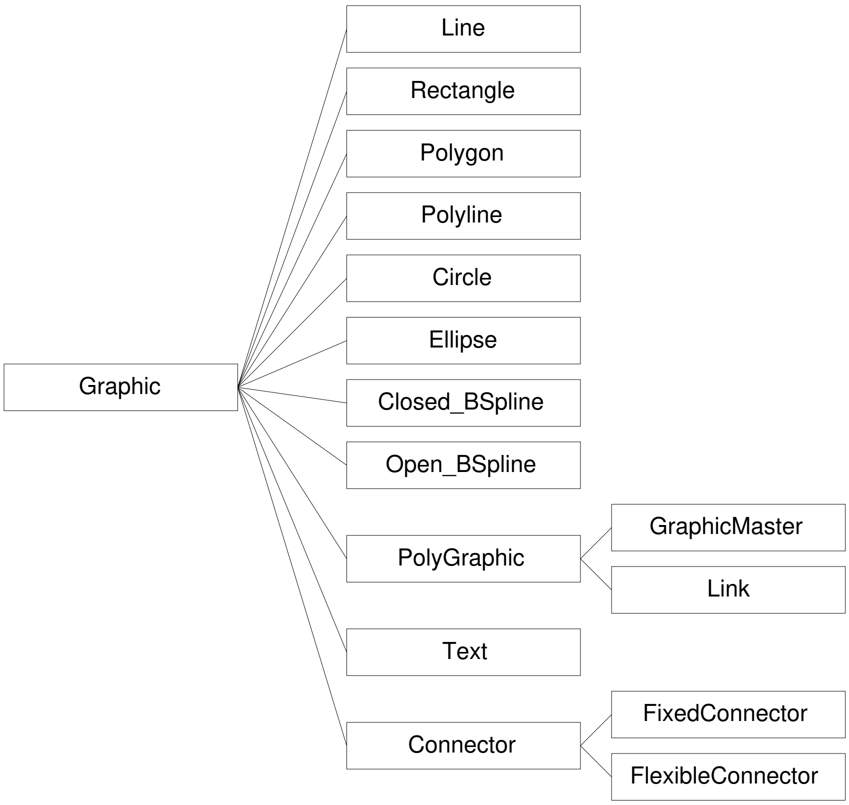

Abbildung 2.1: Die Graphikklassenhierarchie

2.1 Standardgraphiken

Standardgraphiken sind einfache graphische Objekte für geometrische (Grund–) Formen, Texte und Kompositionen von Graphiken.

2.1.1 Die Basisklasse `Graphic`

```
interface Graphic: Glyph
{
    Graphic(Graphic * = nil);

    virtual void drawclipped
    (
```

```
        Canvas *canvas,
        Coord left,Coord bottom,Coord right,Coord top
    );

    virtual Transformer *transformer() const;
    virtual void transformer(Transformer *);
    void eqv_transformer(Transformer &);

    void brush(const Brush *);
    const Brush *brush() const;
    void stroke(const Color *);
    const Color *stroke() const;
    void fill(const Color *);
    const Color *fill() const;
    void font(const Font *);
    const Font *font() const;
    void closed(boolean);
    boolean closed() const;
    void curved(boolean);
    boolean curved() const;
    int ctrlpts(Coord *&x, Coord *&y) const;

    virtual Graphic *parent();
    virtual void parent(Graphic *);
    virtual Graphic *root();

    void translate(Coord x,Coord y);
    void scale(Coord sx,Coord sy,Coord relx = 0.0, Coord rely = 0.0);
    void rotate(float angle,Coord relx = 0.0, Coord rely = 0.0);
    void align(Alignment,Graphic *,Alignment);

    virtual void getbounds(Coord &l,Coord &b,Coord &r,Coord &t);
    virtual void getcenter(Coord &x,Coord &y);
    virtual boolean contains(PointObj &);
    virtual boolean intersects(BoxObj &);

    virtual void append_(Graphic *);
    virtual void prepend_(Graphic *);
    virtual void insert_(GlyphIndex,Graphic *);
    virtual void remove_(GlyphIndex);
    virtual void replace_(GlyphIndex,Graphic *);
    virtual void change_(GlyphIndex);

    virtual GlyphIndex count_() const;
    virtual Graphic *component_(GlyphIndex) const;
    virtual void modified_(GlyphIndex);

    virtual Graphic *first_containing(PointObj &);
    virtual Graphic *last_containing(PointObj &);

    virtual Graphic *first_intersecting(BoxObj &);
    virtual Graphic *last_intersecting(BoxObj &);

    virtual Graphic *first_within(BoxObj &);
    virtual Graphic *last_within(BoxObj &);

    virtual Graphic &operator=(Graphic &);
```

```
    virtual void disconnect_all();
    virtual void update(GlyphViewer * = nil);

    void add_point(Coord x,Coord y);
    void add_curve
    (
        Coord x,Coord y,Coord x1,Coord y1,Coord x2,Coord y2
    );
    void Bspline_move_to
    (
        Coord x,Coord y,Coord x1,Coord y1,Coord x2,Coord y2
    );
    void Bspline_curve_to
    (
        Coord x,Coord y,Coord x1,Coord y1,Coord x2,Coord y2
    );

    virtual void create_handles(Canvas * =nil);
    virtual void draw_handles(Canvas * =nil);
    virtual void delete_handles();

    virtual ClassId get_classid() const;
    virtual boolean is_a(ClassId id) const;
}
```

Abbildung 2.2: Die Schnittstelle zu der Klasse `Graphic`

Eine Graphik ist im Prinzip eine Menge von Koordinatenpaaren (oder Punkten)[1], die einen Linienzug beschreiben. Je nachdem, ob das Attribut `curved` gesetzt oder gelöscht ist, wird die Punktmenge als Bézierkurve oder als Linienzug aufgefaßt. Das Attribut `closed` bestimmt, ob der erste und der letzte Punkt miteinander verbunden werden sollen. (Siehe hierzu auch 1.3).

Dem Konstruktor kann eine initiale Graphik übergeben werden, aus der die Attribute kopiert werden. Mit `drawclipped` wird der Teil der Graphik auf der `Canvas` gezeichnet, der das übergebene Clipping-Rechteck schneidet.

Die Methoden `transformer`, `brush`, `stroke`, `fill`, `font`, `closed` und `curved` erfragen bzw. setzen das entsprechende Attribut . Mit `ctrlpts` können die Koordinaten der Graphik erfragt werden. Der Rückgabewert ist die Anzahl der Koordinatenpaare.

Mit `parent` kann die Eltern-Graphik erfragt bzw. gesetzt werden. Das ist die Graphik, die diese Graphik enthält. Die Methode `root` gibt einen Zeiger auf die Graphik zurück, falls die Graphik keine Eltern-Graphik hat, andernfalls wird von der Eltern-Graphik das Ergebnis von `root` zurückgeliefert.

Die Graphik kann durch `translate`, `scale`, `rotate`, die Winkel im Gradmaß erwartet, und `align` entsprechend deformiert werden, wobei die Parameter `relx` und `rely` einen (absoluten) Bezugspunkt angeben. Mit `align` wird die Graphik zur übergebenen ausgerichtet.

Durch die geometrischen Methoden (siehe Bild 2.3) `getcenter` und `getbounds` können der Mittelpunkt (Punkt `(cx,cy)`) oder das kleinste umgebende Rechteck (mit den Koordinaten `(l,b)` und `(r,t)`) der Graphik bestimmt werden.

Die Methoden `contains` und `intersects` überprüfen, ob die Graphik einen Punkt enthält bzw. ein Rechteck schneidet. Das Ergebnis dieser beiden Operationen wird durch die Füllfarbe beeinflußt (Bild 2.3):

1. Es handelt sich um eine offene Liniengraphik (siehe Graphik 1):
   (a) Wenn das Füllattribut nicht gesetzt ist, liefert `contains` nur dann `true`, wenn der übergebene Punkt auf der Linie liegt. `intersects` ergibt genau dann `true`, wenn das Rechteck die Linie schneidet. In Graphik 1 gilt das nur für den Punkt P2 (`contains`) bzw. nur für Rechteck R3 (`intersects`).

[1] Mit Ausnahme von `Text` und `PolyGraphic`

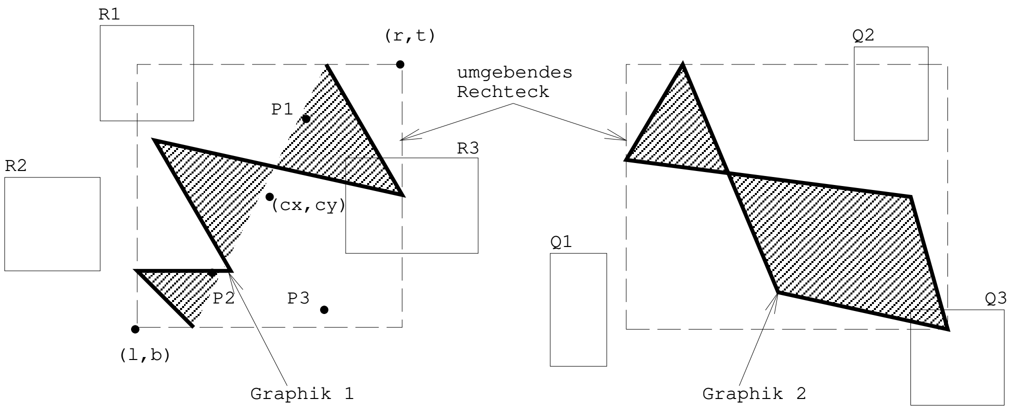

Abbildung 2.3: Die geometrischen Methoden

(b) Bei gesetztem Füllattribut liefert `contains` dann `true`, wenn der übergebene Punkt auf der ausgefüllten Fläche liegt. Das Ergebnis von `intersects` ist dann `true`, wenn das übergebene Rechteck das umgebende der Graphik schneidet. In Graphik 1 liefert `contains` für die Punkte P1, P2 bzw. für alle Punkte, die in der schraffierten Fläche liegen, und `intersects` für die Rechtecke R1 und R3 `true`.

2. Die Liniengraphik ist geschlossen (siehe Graphik 2): Die Methode `contains` liefert genau dann `true`, wenn der übergebene Punkt von der Liniengraphik umschlossen wird. Das Ergebnis von `intersects` ist `true`, wenn das übergebene Rechteck die Graphik schneidet. In Graphik 2 gilt das gesagte für die schraffierte Fläche(`contains`) bzw. für das Rechteck Q3 (`intersects`).

Die Methoden `append_`, ... `modified_` entsprechen den Methoden `append`, ... `modified` in der Klasse `Glyph` (siehe [iv3.1]). Die Definitionen dieser Methoden sind *leer* und nur in der Klasse `PolyGraphic` (siehe 2.1.5) redefiniert, wo sie sinnvoll sind. Graphiken, die weitere Graphiken enthalten, sollten diese Methoden überladen. Obwohl `Graphic` von `Glyph` abgeleitet ist, sollte man nie die von `Glyph` geerbten Methoden `append`, ... `modified` einer Graphik verwenden.

Die Methoden `first_containing`, ... `last_within` sind nur in `PolyGraphic` sinnvoll (definiert). Sie liefern jeweils einen Zeiger auf die erste (`first_*`) bzw. letzte (`last_*`) enthaltene Graphik zurück, die den übergebenen Punkt enthält (`*_containing`), das übergebene Rechteck schneidet (`*_intersecting`) bzw. in dem übergebenen Rechteck enthalten ist (`*_within`).

Mittels `disconnect_all` können alle Verbindungen (siehe 2.2) gelöst werden. Durch `update` werden die Verbindungen aktualisiert. Das können komplexere graphische Operationen sein und sollte immer dann aufgerufen werden, wenn eine Graphik, die einen *Verbinder* enthält, verformt wird. Wird bei `update` ein `GlyphViewer` (siehe 4.1) übergeben, so werden veränderte Bereiche auf der Zeichenfläche (mittels `GlyphViewer::damage`) als beschädigt deklariert.

Durch `add_point` wird der Punkt (x,y) an die an die Punktemenge angeängt, und `add_curve` hängt ein Bézierkurvenstück mit dem Endpunkt (x,y) und den Stützpunkten (x1,y1) und (x2,y2) an. Mit `Bspline_move_to` wird der Endpunkt der Bspline und mit `Bspline_curve_to` die Bspline, die sich aus den Parametern ergibt angehängt. Hierbei ist (x,y) der End- und (x1,y1) bzw. (x2,y2) ein Stützpunkt. Für die genaueren geometrischen Zusammenhänge schaue man bitte in einschlägiger Literatur nach.

Die Methoden `create_handles`, `draw_handles` und `delete_handles` kreieren, zeichnen oder zerstören die Kennzeichner der Graphik. Die Benutzung sollte dem `GraphicMaster` (siehe 2.3) vorbehalten bleiben. Bei eigenen Graphikklassen sollte man ggf. `create_handles` überladen, wenn man andere Kontrollpunkte als in der Basisklasse haben möchte.

Zur Klassenidentifikation dienen die Methoden `get_classid`, die eine (eindeutige) *Kennung* der Klasse zurückliefert. Diese Kennungen sind durch Zahlen realisiert und diese sind für alle Graphiken, Kommandos und Werkzeuge in der Headerdatei `classes.h` definiert. und `is_a`, die genau dann `true` liefert, wenn der Typ des Objektes eine Ableitung bzw. vom Typ der Klasse ist, von dem die übergebene Identifikation ist.

In jeder Ableitung von `Graphic` sollte man `get_classid` und `is_a` redefinieren. Diese neue Klasse sollte dann eine neue, noch nicht vergebene Kennung erhalten, die `get_classid` zurückgibt. Das Ergebnis von `is_a` ist `true`, wenn der Parameter `id` gleich der Kennung ist oder wenn der Aufruf von `<Basisklasse>::is_a` `true` ergibt.

Beispiele: `PolyGraphic` und `Connector` sind von `Graphic` abgeleitet, und `FixedConnector` ist eine Ableitung von `Connector`.

```
Graphic *g1=new Graphic(...);
Graphic *g2=new PolyGraphic(...);
Graphic *g3=new FixedConnector(...);

if(g1->is_a(GRAPHIC))          // wahr, g1 zeigt auf eine Graphic
{...}
if(g1->is_a(POLY_GRAPHIC))     // falsch, g1 zeigt auf eine Graphic
                               // und das ist keine Ableitung
                               // von PolyGraphic
{...}
if(g2->is_a(GRAPHIC))          // wahr, g2 zeigt auf eine PolyGraphic
                               // und das ist eine Ableitung
                               // von Graphic
{...}
if(g2->is_a(CONNECTOR))        // falsch, g2 zeigt auf eine PolyGraphic
{...}
if(g3->is_a(GRAPHIC))          // wahr, g3 zeigt auf einen FixedConnector
                               // und der ist eine indirekte Ableitung
                               // von Graphic
{...}
```

2.1.2 Die Klassen `Line`, `Polyline` und `Polygon`

```
interface Line: Graphic
{
    Line
    (
        const Brush *brush,const Color *stroke,const Color *fill,
        Coord x1,Coord y1,Coord x2,Coord y2,
        Transformer *t=nil
    );
}


interface Polyline: Graphic
{
    Polyline
    (
        const Brush *brush,const Color *stroke,const Color *fill,
        Coord *x,Coord *y,int count,
        Transformer *t=nil
    );
}


interface Polygon: Graphic
{
    Polygon
    (
```

```
        const Brush *brush,const Color *stroke,const Color *fill,
        Coord *x,Coord *y,int count,
        Transformer *t=nil
    );
}
```

Abbildung 2.4: Die Schnittstelle zu den Klassen `Line`, `Polyline` und `Polygon`

Der Konstruktor von `Line` erzeugt eine Linie mit dem Pinsel `brush`, der Rahmenfarbe `stroke`, der Füllfarbe `fill`, den Koordinaten `x1`, `y1` und `x2`, `y2` und der Verformung `t`.

Die Konstrutoren von `Polyline` bzw. `Polygon` arbeiten analog für Polylinien bzw. Vielecke mit dem Unterschied, daß für die Koordinaten Felder `x` und `y` und deren Größe in `count` übergeben werden.

2.1.3 Die Klassen `Circle` und `Ellipse`

```
interface Circle: Graphic
{
    Circle
    (
        const Brush *brush,const Color *stroke,const Color *fill,
        Coord x,Coord y,Coord radius,Transformer *t=nil
    );
};

interface Ellipse: Graphic
{
    Ellipse
    (
        const Brush *brush,const Color *stroke,const Color *fill,
        Coord x,Coord y,Coord radius_x,Coord radius_y,
        Transformer *t=nil
    );
};
```

Abbildung 2.5: Die Schnittstelle zu den Klassen `Circle` und `Ellipse`

Die Konstruktoren dieser beiden Klassen erwarten einen Pinsel (`brush`), eine Rahmen- (`stroke`) und eine Füllfarbe (`fill`), Mittelpunktkoordinaten (`x` und `y`) und einen bzw. zwei Radien (`radius` bzw. `radius_x` und `radius_y`). Ein `Transformer` kann optional übergeben werden.

2.1.4 Die Klassen `Open_BSpline` und `Closed_BSpline`

```
interface Open_BSpline: Graphic
{
    Open_BSpline
    (
        const Brush *brush,const Color *stroke,const Color *fill,
        Coord *x,Coord *y,int count,
        Transformer *t=nil
    );
};
```

```
interface Closed_BSpline: Graphic
{
    Closed_BSpline
    (
        const Brush *brush,const Color *stroke,const Color *fill,
        Coord *x, Coord *y,int count,
        Transformer *t=nil
    );
};
```

Abbildung 2.6: Die Schnittstelle zu den Klassen `Open_BSpline` und `Closed_BSpline`

Die Konstruktoren dieser beiden Klassen erwarten einen Pinsel (`brush`), eine Rahmen- (`stroke`) und eine Füllfarbe (`fill`), die Koordinaten der Stützpunkte (`x` und `y`) und deren Anzahl sowie einen optionalen `Transformer`.

2.1.5 Die Klasse `PolyGraphic`

```
interface PolyGraphic: Graphic
{
    PolyGraphic(Graphic * = nil);
};
```

Abbildung 2.7: Die Schnittstelle zu der Klasse `PolyGraphic`

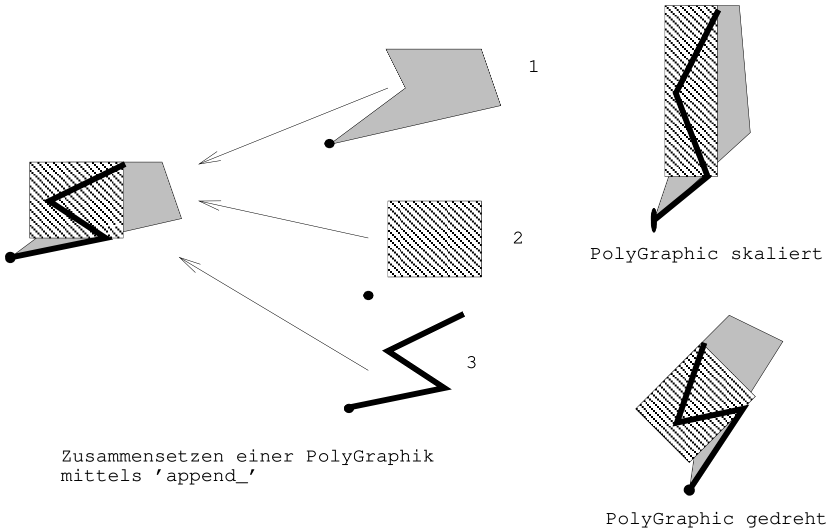

Abbildung 2.8: Beispiel für eine `PolyGraphic`

Eine `PolyGraphic` ist ein *Container* für Graphiken, d.h. daß er beliebig viele Objekte vom Typ `Graphic` enthalten kann. In diesen Container kann mittels der Methoden `append_` (anhängen), `prepend_` (davor einfügen) und `insert_` (wahlfrei) eine beliebige Graphik eingefügt werden. Die enthaltenen Graphiken können durch `remove_` entfernt und durch `replace_` ausgetauscht werden.

Die Elemente der `PolyGraphic` werden in aufsteigender Reihenfolge gezeichnet, daß heißt, das eine Graphik mit einem niedrigeren Index durch eine mit einem höheren Index überdeckt werden kann. Die Koordinaten bzw. der `Transformer` der enthaltenen Graphiken sind immer relativ zur umgebenden `PolyGraphik`. (siehe Bild 2.8)

Dem Konstruktor kann lediglich eine initiale Graphik übergeben werden, deren Attribute (siehe 1.3) die `PolyGraphic` übernimmt.

2.1.6 Die Text-Klasse

```
interface Text: Graphic
{
    Text
    (
        const Font *font,const Color *stroke,const char *str,
        Transformer *t=nil
    );
    virtual void text(const char *);
    virtual const char *text();
};
```

Abbildung 2.9: Die Schnittstelle zur Klasse Text

Dem Konstruktor wird ein Zeichensatz (`font`), eine Zeichenfarbe (`stroke`), ein Zeiger auf den Text (`str`) und ein optionaler `Transformer` übergeben. Mit den Methoden `text` kann der darzustellende Text abgefragt oder gesetzt werden.

2.2 Verbindungen zwischen Graphiken

Graphische Editoren dienen häufig zur Erzeugung von Schaltungsskizzen, Graphen, Bäumen oder Ähnlichem. Diesen Zeichnungen ist gemeinsam, daß sie eine Abstraktion von beispielsweise Schaltungen oder Algorithmen sind. Die abstrakten Objekte, die man bearbeiten möchte, stehen in vielen Fällen in einer Beziehung, so sind Transistoren über Leitungen mit Widerstäden oder Knoten über Kanten mit anderen Knoten verbunden. Diese *Verbindungen* , die nicht notwendig optisch in der Graphik hervortreten müssen[2], unterliegen häufig zusätzlich einer Regelmenge (z.B.: zyklenfreier Graph). Die Verwendung der Klassen dieses Abschnittes ist einem größerem Beispielprogramm (BallsNRays) demonstriert.

2.2.1 Die abstrakte Basisklasse Connector

```
interface Connector: Graphic
{
    virtual void connect(Connector *conn);
    virtual void disconnect(Connector *conn);
    virtual void disconnect(GlyphIndex cindex);
    virtual GlyphIndex c_count() const;
    virtual Connector *connector(GlyphIndex cindex) const;
};
```

Abbildung 2.10: Die Schnittstelle zur Klasse Connector

Ein Verbinder (`Connector`) ist eine Graphik, die eine vorgegebene oder beliebig hohe Anzahl von Verweisen auf andere Verbinder enthält. Die Verbinder *können* einen graphischen Zusammenhang haben, müssen aber nicht.

Mit `connect` wird eine Verbindung zum Verbinder `conn` hergestellt, die mit `disconnect` zerstört wird. Durch `disconnect_all` werden alle Verbindungen, deren Anzahl man mit `c_count` abfragen kann, aufgelöst. Die Methode `connector` gibt einen Verweis auf den Verbinder mit der Nummer `cindex` zurück.

[2] Eine denkbare Anwendung hierfür wäre beispielsweise die Erstellung und Simulation von Ampelanlagen. In dieser Simulation könnte man Regeln der Form: *Immer wenn Ampel A auf rot wechselt soll Ampel B auf rot-gelb wechseln.*

2.2.2 Der FixedConnector und der FlexibleConnector

```
interface FixedConnector: Connector
{
    FixedConnector
    (
        const Brush *brush,const Color *stroke,const Color *fill,
        Transformer *t=nil
    );
};

interface FlexibleConnector: Connector
{
    FlexibleConnector
    (
        const Brush *brush,const Color *stroke,const Color *fill,
        Transformer *t=nil
    );
};
```

Abbildung 2.11: Die Schnittstelle zu den Klassen `FixedConnector` und `FlexibleConnector`

Ein fixierter Verbinder kann mit beliebig vielen flexiblen Verbindern, die ihrerseits nur mit jeweils einem fixierten verbunden sein können, verbunden sein. Die Methode `update` wird von beiden Klassen unterschiedlich ausgewertet:

- Der fixierte Verbinder verändert sich nicht und ruft nur bei den mit ihm verbundenen flexiblen die Methode `update` auf.
- Der flexible hingegen verändert seinen `Transformer`, so daß sein Mittelpunkt genau auf dem des mit ihm verbundenen fixierten Verbinders zu liegen kommt.

Den Konstrutoren beider Klassen übergibt man jeweils einen Pinsel, eine Rahmen- und eine Füllfarbe sowie einen optionalen `Transformer`.

2.2.3 Verbindungen durch Link

```
interface Link: PolyGraphic
{
    Link
    (
        FixedConnector *source=nil,FixedConnector *target=nil,
        const Brush *brush=nil,const Color *stroke=nil,const Color *fill=nil
    );

    Line *get_line() const;
    FlexibleConnector *get_connector(boolean source) const;
    FixedConnector *connector(boolean source) const;
    void connector(FixedConnector *conn,boolean source);
};
```

Abbildung 2.12: Die Schnittstelle zur Klasse `Link`

Eine Verbindungslinie (`Link`) stellt eine Verbindung zwischen zwei fixierten Verbindern — genauer gesagt besteht diese Verbindung aus: `FixedConnector`, `Link` und `FixedConnector` —

graphisch durch eine Linie her. Wird einer der fixierten Verbinder verschoben, so verändert die Linie ihr Aussehen entsprechend.

Die zwei fixierten Verbinder (source und target) werden dem Konstruktor übergeben, sowie ein Pinsel,eine Rahmen- und eine Füllfarbe. Die im Link enthaltene Linie bzw. die enthaltenen flexiblen Verbinder können mit get_line bzw. get_connector erfragt werden. Der Parameter source gibt — wie bei der Methode connector, der die fixierten Verbinder zurückgibt oder setzt — an, ob der Quell- (source==true) oder der Zielverbinder (source==false) gemeint ist.

2.3 Der GraphicMaster

```
class GraphicMaster: PolyGraphic
{
    GraphicMaster(GlyphViewer *gv,Graphic *g=nil,const Color *bg=nil);

    void background(const Color *c);
    const Color* background() const;

    GraphicList *selection() const;
    void get_sbounds(Coord &l,Coord &b,Coord &r,Coord &t) const;

    boolean is_selected(const Graphic *target) const;
    boolean select(Graphic *target,Canvas *canvas=nil,boolean always=false);
    void unselect(Graphic *target);
    void clear_selection();
    void transform_selection(const Transformer *t);

    GlyphViewer *glyphviewer() const;
    void glyphviewer(GlyphViewer *gv);

    void centerpoint(Coord &x,Coord &y) const;
};
```

Abbildung 2.13: Die Schnittstelle zum GraphicMaster

Der GraphicMaster ist die zentrale Klasse zur Verwaltung von Graphiken. Er enthält alle graphischen Objekte, stellt sie dar und ist für die Erneuerung des Hintergrundes zuständig, wenn Objekte verändert wurden. Außerdem können mit ihm Selektionen verwaltet werden.

Der Konstruktor erwartet neben dem zugehörigen GlyphViewer (siehe 4.1) optional eine initiale Graphik und eine Hintergrundfarbe, die mit background gesetzt bzw. abgefragt werden kann.

Die Selektion kann mit selection erfragt werden, wobei GraphicList eine PtrList von **InterViews 3.1** ist. Ob eine Graphic selektiert ist, kann mit is_selected überprüft werden. Sie kann mit select selektiert werden, wobei der Parameter canvas die Zeichenfläche[3] angibt, auf der sich die Graphik befindet. Wird für always false übergeben, so wird der Selektionszustand des Objektes invertiert, andernfalls wird das Objekt immer selektiert. Der Rückgabewert entspricht dem Selektionszustand der Graphik. Mit unselect wird eine Graphik deselektiert und mit clear_selektion wird die Selektion zerstört, das heißt, daß alle selektierten Graphiken deselektiert werden. Ein Transformer kann mit transform_selection auf alle selektierten Objekte angewandt werden.

Mit den Methoden glyphviewer kann der GlyphViewer abgfragt oder gesetzt werden. Durch centerpoint kann man den Mittelpunkt des Ausschnittes ermitteln, den der GraphicMaster im Fenster belegt.

[3] Sollte dieser Parameter nil sein, so wird die Selektion nicht dargestellt

Kapitel 3

Graphiken bearbeiten

Ein Editor soll nicht nur Graphiken darstellen sondern sie auch verändern können, was im allgemeinen auf zwei unterschieliche Arten geschieht:

1. Der Benutzer klickt mit der Maus auf die Zeichenfläche und bewegt den Zeiger mit gedrückter Maustaste. Während der Bewegung verändert ein *Geisterrahmen*, das ist ein (rechteckiger) Rahmen, der genau die (zu verändernde) Graphiken umschließt oder eine Linie, sein Aussehen entsprechend. Wenn die Maustaste losgelassen wird, wird die Graphik entsprechend verändert.

2. Der Benutzer gibt mittels eines Dialoges[1] an, wie sich die (ausgewählte) Graphik verändern soll. Wird der Dialog akzeptiert, so wird die Graphik entsprechend verändert.

Die Hilfsmittel (bzw. Klassen), die die Graphiken nach dem ersten Typ bearbeiten, wollen wir *Werkzeuge* nennen und die des zweiten Typs *Kommandos* nennen.

3.1 Werkzeuge

Ein Werkzeug ist besonders dann hilfreich, wenn der Benutzer sehen soll, was er macht. So können beispielsweise Graphiken mit Werkzeugen nach anderen Graphiken ausgerichtet werden.

3.1.1 Die Basisklasse `Tool`

```
interface Tool: Resource
{
    virtual boolean grasp(const Event &) = 0;
    virtual boolean manipulating(const Event &) = 0;
    virtual boolean effect(const Event &) = 0;

    virtual boolean button_used(unsigned);

    GraphicMaster *graphic_master();
    void graphic_master(GraphicMaster *);

    virtual ClassId get_classid() const;
    virtual boolean is_a(ClassId) const;
}
```

Abbildung 3.1: Die Schnittstelle zur Klasse `Tool`

[1] Ein Dialog könnte in diesem Fall sowohl eine *Dialogbox*, ein Menüeintrag als auch eine andere interaktive Eingabemöglichkeit (von **InterViews 3.1**) sein

Die Methode `grasp` wird (direkt) nach dem Drücken der Maustaste aufgerufen und liefert als Ergebnis, ob `manipulating` aufgerufen werden soll. Dies wird dann in einer Schleife sooft aufgerufen bis es `false` zurückgibt. Der Rückgabewert von `effect` gibt an, ob die Zeichenfläche verändert wurde.

Die Methode `button_used` gibt an, welcher Mausknopf *zusätzlich* zum linken gebraucht wird. Gültige Werte sind `GlyphViewer::left`, `GlyphViewer::middle` und `GlyphViewer::right`. Für den Wert `GlyphViewer::left` sollte immer `true` zurückgegeben werden.

Mit `graphic_master` kann der `GraphicMaster` abgefragt oder gesetzt werden. Für die Methoden `get_classid` und `is_a` gilt dasselbe wie in 2.1.1.

3.1.2 Graphiken einfügen mit dem `GraphicTool`

```
interface GraphicTool: Tool
{
    GraphicTool(GraphicMaster *,Graphic *);
    Graphic *prototype() const;
};
```

Abbildung 3.2: Die Schnittstelle zur Klasse `GraphicTool`

Das `GraphicTool` dient zum Einfügen einer beliebigen, bei der Konstruktion fest vorgegebenen Graphik, die mit `prototype` abgfragt werden kann.

Beispiel: Wir wollen mit der rechten Maustaste Graphiken vom Typ `MyGraphic` in den BezGraphicMaster einfügen.

```
Session *session=new Session(...);
GlyphViewer *glyphviewer=new GlyphViewer(...);
GlyphMaster *graphicmaster=glyphviewer->root();
MyGraphic *prototype=new MyGraphic(...);
GraphicTool *graphictool=new GraphicTool(graphicmaster,prototype);
Glyph *glyph;

// Bilde einen Editor aus InterViews-Elementen
// und fuege darin 'glyphviewer' ein
glyphviewer->tool(graphictool,GlyphViewer::right);
// ...
session->run_window(new ApplicationWindow(glyph));
```

Jedesmal, wenn mit der rechten Maustaste in den `GlyphViewer` geklickt wird, wird ein Geisterrahmen erzeugt, der solange die Taste gedrückt bleibt verschoben werden kann. Wird die Taste losgelassen, wird an der Position des Geisterrahmens eine mittels `MyGraphic::clone()` erzeugte Kopie von `prototype` eigefügt.

3.1.3 Das `SelectTool` wählt aus

```
interface SelectTool: Tool
{
    SelectTool(GraphicMaster *);
};
```

Abbildung 3.3: Die Schnittstelle zur Klasse `SelectTool`

Mit dem `SelectTool` kann die Selektionsliste des `GraphicMasters` geändert werden. Wird eine Graphik mit der Maus angeklickt, so wird die Selektion gelöscht und die angeklickte Graphik selektiert. Falls die SHIFT-Taste gedrückt wird, wird die Selektion nicht gelöscht und der

Selektionszustand der angeklickten Graphik invertiert. Wenn man auf die leere Zeichenfläche klickt, kann man mit gedrückter Maustaste ein Selektionsrechteck aufziehen. Es werden nach dem Loslassen der Maustaste alle Graphiken selektiert, die das Selektionsrechteck schneiden.

3.1.4 Das `MoveTool` zum verschieben

```
interface MoveTool: Tool
{
    MoveTool(GraphicMaster *);
};
```

Abbildung 3.4: Die Schnittstelle zur Klasse `MoveTool`

Mit dem `MoveTool` kann sowohl die Selektionsliste des `GraphicMasters` geändert als auch Graphiken verschoben werden. Der Unterschied zur Semantik des `SelectTools` besteht darin, daß eine Graphik verschoben werden kann, wenn der Mausknopf gedrückt bleibt. Sollten mehrere Graphiken selektiert sein und man klickt eine davon an, dann kann die ganze Selektion (mit gedrückter Maustaste) verschoben werden.

3.1.5 Das `MenuTool`

```
interface ToolMenu: Resource
{
    virtual Menu *menu();

    void target(Graphic *);
    Graphic *target();

    static void main_window(Window *);
    static Window *main_window();
};


interface MenuTool: Tool
{
    MenuTool(GraphicMaster *,ToolMenu *);
};
```

Abbildung 3.5: Die Schnittstelle zu den Klassen `ToolMenu` und `MenuTool`

Das `MenuTool` verwaltet zu Graphiken ein Menü von **InterViews 3.1** mit Hilfe der Klasse `ToolMenu`. Dieses Menü, das wir von nun an *Popup-Menü* nennen, kann mit der Methode `menu` abgefragt werden und mit `target` kann man die ausgewählte Graphik setzen oder abfragen. Die statischen Methoden `main_window` setzen[2] bzw. erfragen das Hauptfenster der Applikation.

Beispiel: Wir wollen ein Popup-Menü mit den Einträgen: *Info...*, *Edit...* und *Duplicate* erzeugen.

```
class MyMenu: public ToolMenu
{
    public:
        MyMenu(): ToolMenu(popup()) {}

        // Callback-Funktionen
        void info() {...}
```

[2] Das muß man zu Anfang des Programms machen.

```
        void edit() {...}
        void duplicate();
    private:
        Menu *popup() const;
};

void MyMenu::duplicate()
{
    Graphic *g=(Graphic *) target()->clone();
    GraphicMaster *gm=(GraphicMaster *) target()->root();
    Coord l,b,r,t;

    // Damit die neue Graphik nicht genau auf der alten liegt
    g->getbounds(l,b,r,t);
    g->translate((r-l)/2,(t-b)/2);
    // kopierte Graphik einfuegen
    gm->append_(g);
}

declareActionCallback(MyMenu)
implementActionCallback(MyMenu)

Menu *MyMenu::popup() const
{
    WidgetKit &kit=*WidgetKit::instance();
    LayoutKit &layout=*LayoutKit::instance();
    Menu *menu=kit.pulldown();
    MenuItem *item;
    Glyph *g;

    g=layout.rmargin(kit.label("Info..."),0.0,fil,0.0);
    item=kit.menu_item(g);
    item->action(new ActionCallback(MyMenu)(this,&MyMenu::move));
    menu->append_item(item);
    g=layout.rmargin(kit.label("Edit..."),0.0,fil,0.0);
    item=kit.menu_item(g);
    item->action(new ActionCallback(MyMenu)(this,&MyMenu::edit));
    menu->append_item(item);
    g=layout.rmargin(kit.label("Duplicate"),0.0,fil,0.0);
    item=kit.menu_item(g);
    item->action(new ActionCallback(MyMenu)(this,&MyMenu::duplicate));
    menu->append_item(item);
    return menu;
}
```

Die Belegung einer Maustaste mit einem `MenuTool` erfolgt analog zum Beispiel in 3.1.2.

3.1.6 Hier `ConnectTool` ich verbinde

```
interface ConnectTool: Tool
{
    ConnectTool(GraphicMaster *);
}
```

Abbildung 3.6: Die Schnittstelle zur Klasse `ConnectTool`

Dieses Werkzeug dient zur Verbindung von *beliebigen* Graphiken, hat aber nach Ausführung *keine* Auswirkung auf diese Graphiken. Klickt man eine Graphik an, dann wird zwischen diesem (angeklickten) Punkt und dem Mauszeiger eine Linie — oder auch *Geisterlinie* — gezogen, solange die Maustaste gedrückt bleibt. Wenn die Taste losgelassen wird, verschwindet die Linie vom Bildschirm.

Will man zum Beispiel Verbinder mit einem `Link` verbinden, so sollte man von diesem `Tool` ableiten und die Methode `effect` überladen. Die Graphiken, zwischen denen die Linie gezogen wurde, können mit der geschützten Methode `connection` und die Endpunkte können mit `line` (geschützt) abgefragt werden. Diese beiden Methoden liefern einen booleschen Rückgabewert, der genau dann wahr ist, wenn zwei Graphiken miteinander verbunden wurden.

Beispiel:

```
// Verwende als Klassenkennung eine Zahl,
// die an noch keine andere Klasse vergeben wurde.
#define MY_CONNECT_TOOL 2500

class MyConnectTool: public ConnectTool
{
    MyConnectTool(GraphicMaster *gm): ConnectTool(gm){...}

    virtual boolean effect(const Event &);

    virtual ClassId get_classid() const;
    virtual boolean is_a(ClassId) const;
}

boolean MyConnectTool::effect(const Event &e)
{
    Graphic *g1,*g2;

    // Als erstes den Event auswerten, Geisterlinie loeschen usw.:
    ConnectTool::effect(e);
    // Test auf Verbindung:
    if(connection(g1,g2))
    {
        // Die Graphiken g1 und g2 wurden verbunden.
        // diese Verbindung kann ausgewertet werden.
        ...
        return true; // Nur bei Erfolg!!!
    }
    return false;
}

ClassId MyConnectTool::get_classid() const
{
    return MY_CONNECT_TOOL;
}

boolean MyConnectTool::is_a(ClassId id) const
{
    return id==MY_CONNECT_TOOL || ConnectTool::is_a(id);
}
```

3.1.7 `NOPTool` – das macht doch nichts

```
interface NOPTool: Tool
{
   NOPTool(GraphicMaster *);
```

```
};
```

Abbildung 3.7: Die Schnittstelle zur Klasse `NOPTool`

Das `NOPTool` dient als Dummy, wenn man kein Werkzeug im `GlyphViewer` einstellen möchte, d.h. man muß eine Instanz dieser Klasse auf die linke Maustaste — mit `GlyphViewer::tool` — legen.

3.2 Kommandos

Ein Kommando wirkt auf alle selektierten Objekte des `GraphicMasters` und kann durch Menüpunkte, Dialoge usw. aufgerufen werden.

3.2.1 Die Basisklasse `Command`

```
interface Command: Resource
{
    virtual void execute();

    GraphicMaster *graphic_master() const;

    virtual ClassId get_classid() const;
    virtual boolean is_a(ClassId) const;
};


interface CommandAction: Action
{
    CommandAction(Command *);

    virtual void execute();
};
```

Abbildung 3.8: Die Schnittstelle zur Klasse `Command`

Die Methode `execute` führt das Kommando aus und `graphic_master` gibt den zugehörigen `GraphicMaster` zurück. Für die Methoden `get_classid` und `is_a` gilt dasselbe wie in 2.1.1.

Mit einer `CommandAction` kann ein Kommando direkt an ein `MenuItem` oder einen `Button` gehängt werden, ohne eine spezielle `ActionCallback` benutzen zu müssen.

3.2.2 Löschen mit dem `CutCmd`

```
interface CutCmd: Command
{
    CutCmd(GraphicMaster *);
};
```

Abbildung 3.9: Die Schnittstelle zur Klasse `CutCmd`

Das `CutCmd` löscht alle selektierten Graphiken aus dem `GraphicMaster`, der als Parameter im Konstruktor übergeben werden muß.

3.2.3 Das `MoveCmd` verschiebt

```
interface MoveCmd: Command
{
    MoveCmd(GraphicMaster *,Coord,Coord);

    void set_movement(Coord,Coord);
    void get_movement(Coord &,Coord &) const;
};
```

Abbildung 3.10: Die Schnittstelle zur Klasse `MoveCmd`

Mit diesem Kommando kann eine Selektion um die Koordinaten, die man im Konstruktor neben dem `GraphicMaster` übergibt, verschieben. Mit `set_movement` kann man die Verschiebung setzen und mit `get_movement` abfragen.

Kapitel 4

Weitere Klassen

4.1 Der GlyphViewer

```
interface GlyphViewer: InputHandler, Adjustable
{
    enum {left,middle,right};

    GlyphViewer
    (
        Coord stdw=-1.0,Coord stdh=-1.0,
        Coord pagew=1600,Coord pageh=1600
    );

    virtual GraphicMaster *root() const;
    virtual void root(GraphicMaster *);

    virtual Tool *tool(unsigned button=GlyphViewer::left) const;
    virtual void tool(Tool *,unsigned button=GlyphViewer::left);

    void damage(Coord l,Coord b,Coord r,Coord t);
    void damage_all();
    void repair();

    void panner(Slider *s);
    Slider *panner() const;

    Adjustable *zoomer() const;
};
```

Abbildung 4.1: Die Schnittstelle zur Klasse `GlyphViewer`

Der `GlyphViewer` dient als Schnittstelle zwischen **InterViews 3.1** und der Graphikbibliothek. Er stellt einen Ausschnitt des `GraphicMasters`, der bei Konstruktion automatisch erzeugt wird, (siehe 2.3) dar, verwaltet `Slider`, die den Ausschnitt oder die Größe des `GraphicMaster` verändern und behandelt (Tastatur- und) Mauseingaben.

Der Konstruktor erwartet neben einer Standardgröße (`stdw` und `stdh`) die Seitengröße des `GraphicMasters`, der mit den Methoden `root` abgefragt oder gesetzt werden kann. Das aktuelle Werkzeug (siehe 3.1) kann mit `tool` erfragt oder gesetzt werden, wobei der optionale Parameter die Maustaste angibt. Sollten die mittlere oder rechte Maustaste unbelegt sein, so wird dafür das Werkzeug der linken Maustaste verwendet.

Mit `damage` bzw. `damage_all` können Bereiche bzw. der ganze Bereich des `GraphicMasters` als ungültig gekennzeichnet werden. Die Methode `repair` zeichnet ungültige Bereiche neu.

Die Methoden `panner` setzen bzw. erfragen den (`XY-`)`Slider`, der für die Verschiebung der `GraphicMasters` zuständig ist. Der `Adjustable zoomer` ist für die Vergrößerung des `GraphicMasters` zuständig.

4.2 Die Klasse `Handles`

`Handles` (*Kennzeichener*) dienen zum Anzeigen von Selektionen. Sie bestehen aus einer Menge von kleinen Quadraten (*Kontrollpunkte*), die die Graphik in einer festvorgegebenen Weise umrahmen. Bei Liniengraphiken bestehen die `Handles` aus zwei Kontrollpunkten, die sich an den Enden der Linie befinden, und bei allen anderen Graphiken sind es acht Kontrollpunkte, die im Rechteck[1] angeordnet sind. Dabei sind vier Kontrollpunkte an den Eck- und die anderen vier an den Seitenmittelpunkten angeordnet.

Alle Graphikklassen aus **TinIV** erzeugen ihre Kennzeichner selbst (siehe dazu die Methode `create_handles` in 2.1.1).

```
interface Handles
{
    Handles(const Coord *x,const Coord *y,int count,Canvas *c=nil);
    virtual ~Handles();

    void draw(Canvas * =nil);
    void redraw();
    void erase();
    void getbounds(Coord &,Coord &,Coord &,Coord &);
    boolean drawn();
    static Coord tol();

    int get_original(Coord *&,Coord *&);
};
```

Abbildung 4.2: Die Schnittstelle zur Klasse `Handles`

Der Konstruktor erwartet neben den Koordinaten und deren Anzahl auch die `Canvas`, auf der sich die zugehörige Graphik befindet. Mit `draw` werden die Kontrollpunkte gezeichnet, mit `redraw` neugezeichnet und mit `erase` gelöscht. Mit der Methode `drawn` kann man abfragen, ob die `Handles` gezeichnet wurden und `getbounds` liefert den Umfang. Die statische Methode `tol` gibt die halbe Breite eines Kontrollpunktes zurück und `get_original` liefert neben den Koordinatenmittelpunkten auch deren Anzahl zurück.

4.3 Graphiken und `Glyphs`

Ein graphischer Editor soll eine möglichst intuitive Oberfläche besitzen, das heißt, daß das Programm möglichst ohne ein Handbuch bedienbar ist. Bilder (*Ikonen*) erleichtern im allgemeinen die Verständlichkeit eines Programmes[2]. So ist es beispielsweise verständlicher, wenn auf dem Knopf zur Generierung von Dreiecken ein Dreieck gemalt ist, als wenn eben auf diesem Knopf das Wort *Dreieck* steht. Oder eine Dialogbox die ein rotes Fragezeichen enthält, suggeriert dem Benutzer *ohne* Text, daß von ihm eine Entscheidung erwartet wird. Aus diesem Grund unterstützt **TinIV** Ikonen in beliebigen `Glyphs`.

[1] Hier ist das kleinste (waagerechte) Rechteck gemeint, das die Graphik umschließt.

[2] Sie sollten allerdings auch nicht zu verschwenderisch oder gekünstelt eingesetzt werden. So wirken zu viele Ikonen verwirrend, und bestimmte Sachverhalte lassen sich einfacher durch einen (kurzen) Text ausdrücken.

4.3.1 Die Klasse GraphicGlyph

```
class GraphicGlyph: public MonoGlyph
{
    GraphicGlyph(Graphic *,const Color *active=nil);

    void state(TelltaleState *);
    TelltaleState *state() const;
};
```

Abbildung 4.3: Die Schnittstelle zu der Klasse `GraphicGlyph`

Der Konstruktor erwartet neben der darzustellenden Graphik eine optionale Farbe, die als Füllfarbe gesetzt wird, wenn vom `TelltaleState`, der mit `state` gesetzt bzw. abgefragt werden kann, das Flag `is_chosen` gesetzt ist. Wird dem Konstruktor keine Farbe übergeben oder wird kein `TelltaleState` gesetzt, dann ändert die Graphik ihre Farbe nicht.

4.3.2 Spezielle Knöpfe für Werkzeuge

Häufig möchte man die Werkzeuge in Knöpfen[3] neben der Zeichenfläche angeordnet haben und das aktuelle durch anklicken auswählen. Die Semantik dieser Knöpfe ist einer von vielen, was denen aus dem `WidgetKit` entspricht, die man mit den Methoden `palette_button` oder `radio_button` erhält. Das Aussehen dieser Knöpfe sollte aber denen entsprechen, die man mit `push_button` erhält. Die folgende Funktion stellt solche Knöpfe bereit.

Da es umständlich ist, für jedes Werkzeug mit den Pseudo-Templates `ActionCallbacks` zu erzeugen, wird noch eine spezielle `ToolAction` bereitgestellt.

```
Button *tool_button(Glyph *g,TelltaleFlags f,TelltaleGroup *tg,Action *a);

class ToolAction: public Action
{
    ToolAction(Tool *);
};
```

Abbildung 4.4: Die Schnittstelle zu der Funktion `tool_button` und der Klasse `ToolAction`

Die Funktion `tool_button` erwartet neben dem darzustellenden inneren `Glyph` die möglichen `TelltaleFlags`, die Gruppe (`TelltaleGroup`) des Knopfes und eine `Action`, die bei Selektion ausgeführt wird.

Der Konstruktor von `ToolAction` erwartet ein `Tool`, das im `GlyphViewer` auf die linke Maustaste gelegt wird, wenn die Aktion ausgeführt wird.

4.4 Das Farbenverzeichnis

Farben (Klasse: `Color`) sind die wahrscheinlich häufigst benutzten Resourcen einer graphischen Applikation. Besonders *Standardfarben* , wie Weiß, Schwarz oder die Primär- und Sekundärfarben werden sehr häufig als Schmuckfarben benötigt.

Da die Klasse `Color` von `Resource` abgeleitet ist, ist eine Wiederverwendung prinzipiell möglich, aber ohne weitere Hilfsklassen *relativ unpraktisch.* Außerdem wäre es schön, Farben durch symbolische Namen – beispielsweise den Farbennamen oder den Hauptverwendungszweck – anzusprechen.

[3] in deutsch: Buttons

```
interface ColorDictionary: Resource
{
    ColorDictionary();

    static const Color *color
    (
        ColorIntensity r,ColorIntensity g,ColorIntensity b,
        float alpha=1.0,const char *name=""
    );

    static const Color *color(const char *);
    static const Color *color(const String &);
};
```

Abbildung 4.5: Die Schnittstelle zur Klasse `ColorDictionary`

Da alle Methoden der Klasse `static` sind, ist der Konstruktoraufruf aus der Anwendung heraus unnötig. Wenn in den Elemenstfunktionen nach einem Namen gesucht wird, dann wird immer zuerst in der Tabelle und danach in den X-Tabellen gesucht. Sollte der Eintrag nur in der X-Tabelle vorhanden sein, dann wird ein entsprechender Eintrag in der Tabelle angelegt.

Mit der Methode `color(ColorIntensity, ColorIntensity, ColorIntensity, float, const char *)` kann eine Farbe nach ihren Farbwerten gesucht werden, sollte die Farbe nicht gefunden werden, so wird nach einem Eintrag mit dem Namen `name` gesucht, und falls der auch nicht gefunden wird, so wird ein neuer Eintrag erzeugt, der den übergebenen Namen erhält. Falls die Farbe unter diesen Werten unter einem anderen Namen eingetragen sein sollte, so wird ein neuer Eintrag mit dem übergebenen Namen angelegt. Es gelten also folgende Prioritäten:

1. Suche nach den Farbwerten in der Tabelle.
2. Suche nach dem Namen in der Tabelle.
3. Suche in der X-Tabelle.
4. Anlegen eines neuen Eintrages mit den Farbwerten und dem Namen.

Es wird *immer* eine Farbe zurückgeliefert.

Eine Farbe kann mit `color(const char *)` bzw. `color(const String &)` nach ihrem symbolischen Namen gesucht werden. Sollte kein entsprechender Eintrag gefunden werden, dann wird `nil` zurückgegeben.

Anhang A

Erzeugung der Bibliothek und Installation

Zur Installation sind folgende Schritte auszuführen:

1. Entpacken der Datei `TinIV.tar.Z` in einem beliebigen Ordner. Es wird ein neuer Ordner **TinIV** mit Unterverzeichnissen und Dateien angelegt.

2. In das Verzeichnis `TinIV/installing` wechseln und die Datei `Makefile.glob` editieren. Die drei Variablen sind an die jeweilige Umgebung anzupassen:

   - Die Variable `IVDIR` ist auf das Rootverzeichnis von **InterViews 3.1** zu setzen.
   - Das Verzeichnis, in dem die Dateien von **TinIV** stehen, ist der Variablen `TINIVDIR` zu übergeben.
   - Das Verzeichnis, in dem die `X11`–Bibliotheken stehen, wird der Variablen `X11DIR` zugewiesen.

3. Setzen der Shellvariablen `CPU`. Also beispielsweise `setenv CPU MIPSEL` für die `DEC-MIPSEL`–Architektur.

4. Das Kommando `make World` eingeben. Jetzt werden verschiedene Dateien (u.a. die Bibliothek und ein Beispielprogramm) erzeugt, was etwas Zeit beansprucht.

Anhang B

Einbindung in eigene Programme

Folgende Zeile sollte in das `Imakefile` ($\neq$ `Graphic/Imakefile`) des Programms nach der Zeile: `#ifdef InObjectCodeDir` eingefügt werden:

```
APP_CCLDLIBS = -lGraphic
```

Das Makefile sollte mit `timkmf -a` und *nicht* mit `ivmkmf -a` erzeugt werden. Die Bibliothek wird dann an der richtigen Stelle eingebunden. Das Programm `timkmf` befindet sich im Verzeichnis `TinIV/bin/<cpu>`, wobei `<cpu>` von der jeweiligen Rechnerarchitektur abhängt.

WICHTIG: Es ist darauf zu achten, daß die Graphik-Bibliothek vor der Mathematik- (`libm.a` bzw. `-lm`) und vor der `InterViews`-Bibliothek (`libIV.a` bzw. -lIV) eingebunden wird.

Literaturverzeichnis

[iv3.1] M. Linton u.a.; InterViews Reference Manual, Version 3.1; Dec. 14, 1992

[strou] B. Stroustrup; Die C++-Programmiersprache, 2. Auflage; Addison-Wesley

Index