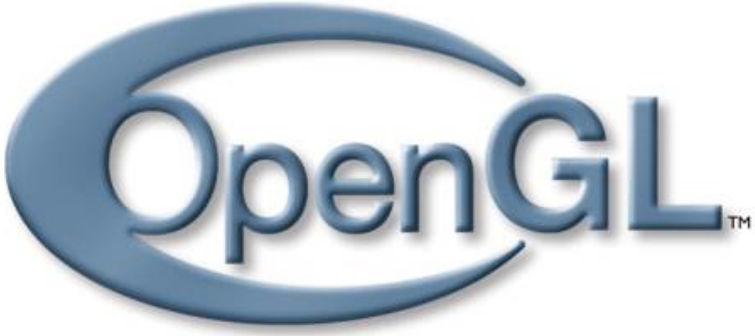
Bremen

CG
VR

Computer-Graphik
Einführung in OpenGL



G. Zachmann

University of Bremen, Germany

cgvr.cs.uni-bremen.de

Hintergrund und Geschichte

- ... auf der Suche nach einer einheitlichen Software-Schnittstelle (API: Application Programming Interface) zur Programmierung von Graphiksystemen
- Standardisierungsbemühungen
  - GKS, PHIGS, ...
- „Proprietäre Systeme“
  - HP: Starbase, SGI: GL (Graphics Library)
- Gewinner: SGI mit GL in Verbindung mit sehr guter Hardware
- OpenGL (1992, Mark Segal & Kurt Akeley)
- Konkurrenz nur noch durch Microsoft (Direct3D)

OpenGL

- OpenGL ist ein Software-Interface für Graphik-Hardware mit ca. 250 verschiedenen Kommandos
  - Hardware-unabhängig
- Warum „Open“?
  - offen für Lizenznehmer
  - verwaltet vom Architecture Review Board (ARB)
    - NVIDIA, ATI, IBM, Intel, SGI, ...
  - von jedem Lizenznehmer erweiterbar (Extension)
- Nicht dabei:
  - Handhabung von Fenstern/Windows
  - Benutzereingabe

Warum OpenGL

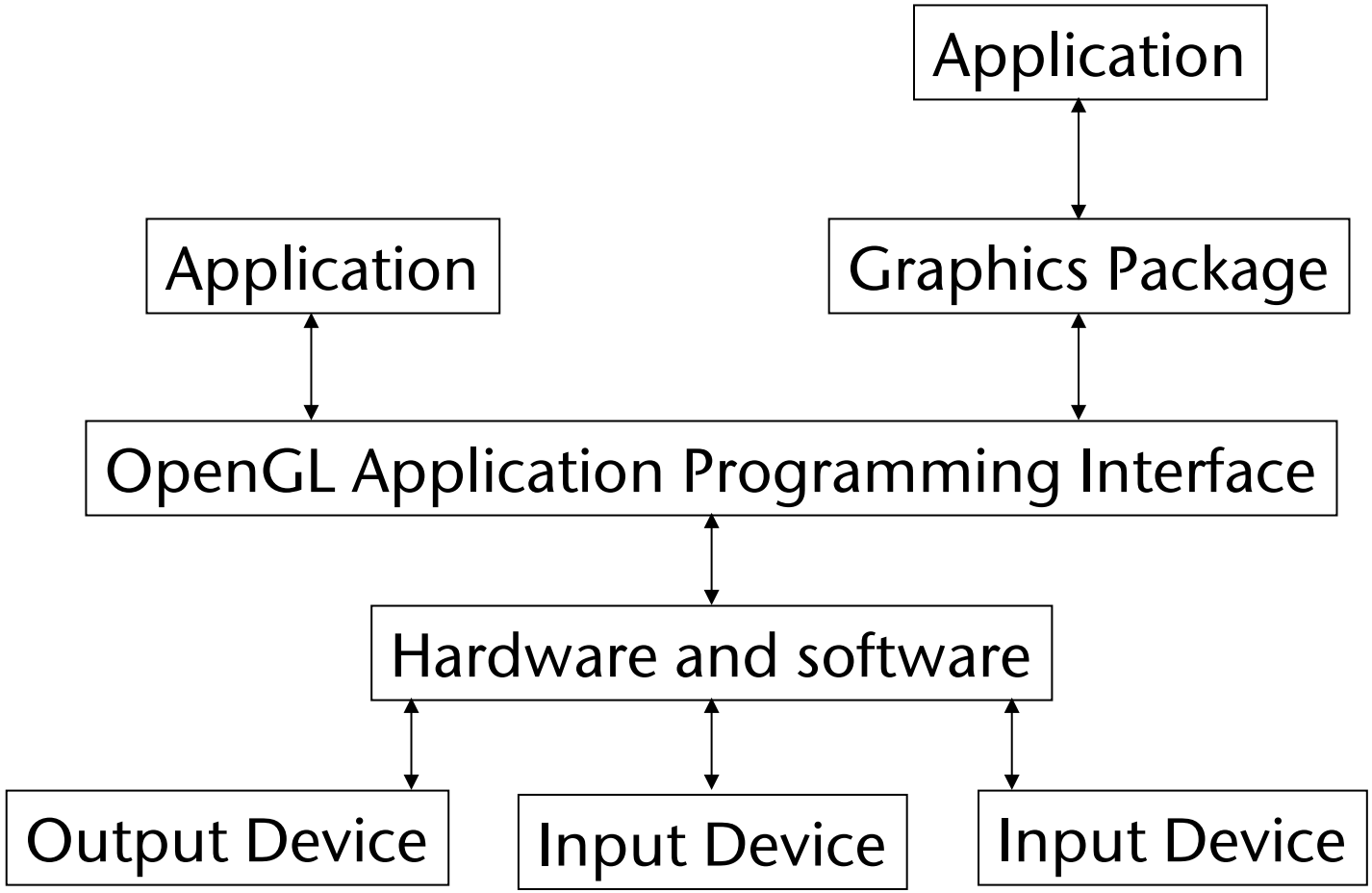
- Standard für Rendering von 3D Graphiken
  - Implementierung als C/C++ Bibliothek von diversen Herstellern:

    nVIDIA, ATI, SiliconGraphics, Microsoft, [*Mesa*]
  - Enthalten in jedem Windows-, MacOS-, Linux- und Unix-System
- Plattform-unabhängig
- Hardware-unterstützt
- Schnell & einfach
- Unabhängig vom Window-Manager
- Wird voraussichtlich der Standard im Handheld-Markt (OpenGL ES)
- Viele weitere offene Standards der Khronos-Group

Einführung

- OpenGL Core: Basisprimitive (Punkte, Linien, Polygone...)
- Darauf aufbauend gibt es diverse Tools:
  - OpenGL Utility Library (GLU): standardmäßig dabei für Oberflächen (Quadrics, NURBS...)
- OpenGL ist eine „State-Machine“
  - Man versetzt die „Maschine“ in einen Zustand, der so lange besteht, bis er wieder verändert wird
  - Beispiel: ab jetzt alles rot, ab jetzt dieses Material, ab jetzt diese Transformation
  - Effizienter, als Daten jedes Mal neu zu übergeben

Programm-Übersicht

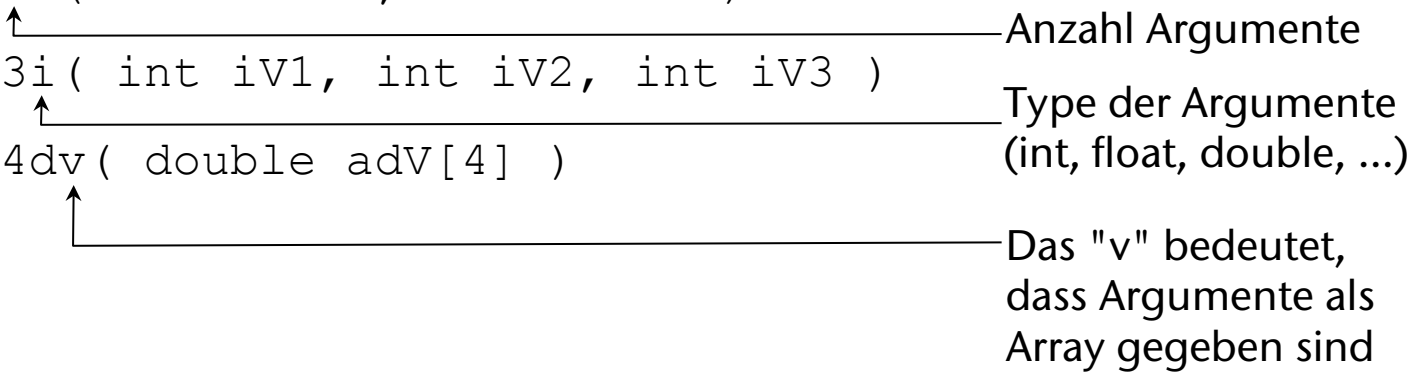


Die Grundstruktur von OpenGL

- Low-Level-API
  - Hardware-nah aber Hardware-unabhängig
- 2 Arten von Funktionen
  - Zustand ändern
  - Primitive darstellen
- Ein reines *immediate mode* System (zumindest früher):
  - Sehr einfache Befehle
  - Direktes Durchreichen an die HW
  - Dreiecks-basiert, keine interne Repräsentation der Szene
- Klarer Namensraum
  - Befehle fangen mit `gl...` an
  - Konstanten mit `GL_...`

Geometrische Primitive in OpenGL

- Alle geometrischen Primitive werden durch ihre Eckpunkte beschrieben → OpenGL-Befehl: `glVertex()`
- Die Bedeutung der Suffixe:
  - `glVertex2f( float fV1, float fV2 )`
  - `glVertex3i( int iV1, int iV2, int iV3 )`
  - `glVertex4dv( double adV[4] )`



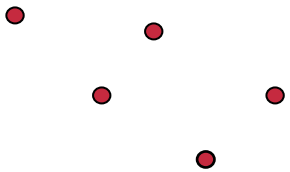
- Beispiele:

```
glVertex2s( 2, 3 );
glVertex3d( 0.0, 0.0, 3.1415926535898 );
glVertex4f( 2.3, 1.0, -2.2, 2.0 );

Gldouble ad_vect[3] = {5.0, 9.0, 1992.0};
glVertex3dv( ad_vect );
```

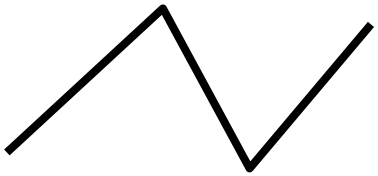
Prinzipielle Vorgehensweise des Zeichnens

- Einfügen der Eckpunkte zwischen **`glBegin()`** … **`glEnd()`**
  - Kann beliebigen C Code enthalten
  - Beinhaltet Befehle wie **`glVertex3f, glColor3f`** (= Attribute der Vertices)
  - Keine sonstigen OpenGL-Befehle
  - Attribute müssen gesetzt sein, bevor die Koordinaten eines Vertex abgeschickt werden!
- Client-Server Modell:
  - Client (= App.) erzeugt Eckpunkte, Server (= OpenGL + Hardware) zeichnet; selbst wenn beides auf dem selben Rechner läuft
  - Dazwischen ein Buffer
  - **`glFlush()`** & **`glFinish()`**
    melden das Ende eines Frames (=Bildes)

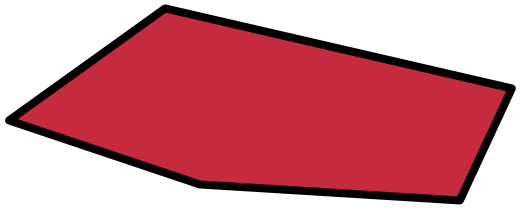
Primitive in OpenGL



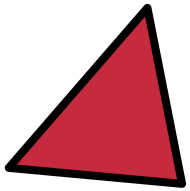
Punkte



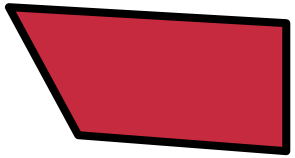
Linien



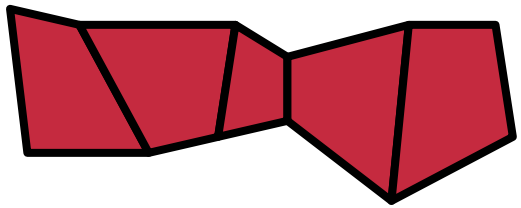
Polygone



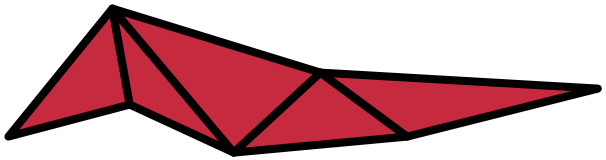
Dreiecke



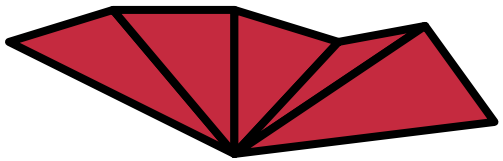
Vierecke
(*Quad* / *Quadrangle*)



Band aus Vierecken
(*Quad Strip*)



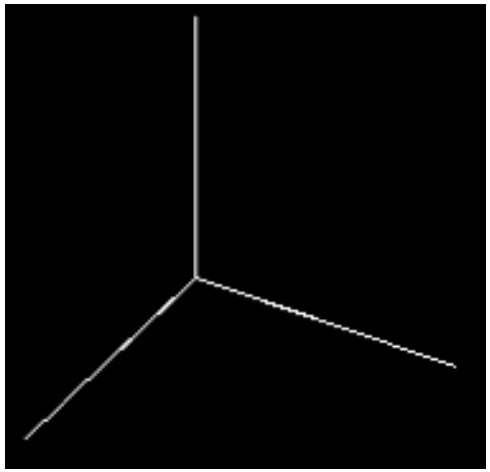
Band aus Dreiecken
(*Triangle Strip*)



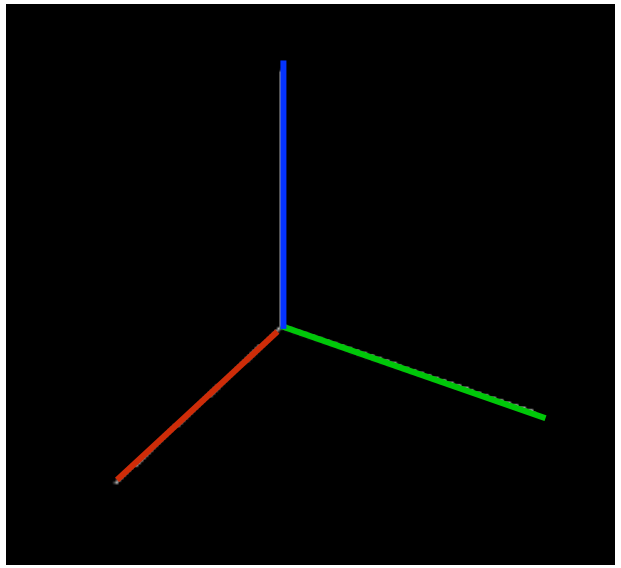
Fächer aus Dreiecken
(*Triangle Fan*)

Beispiel

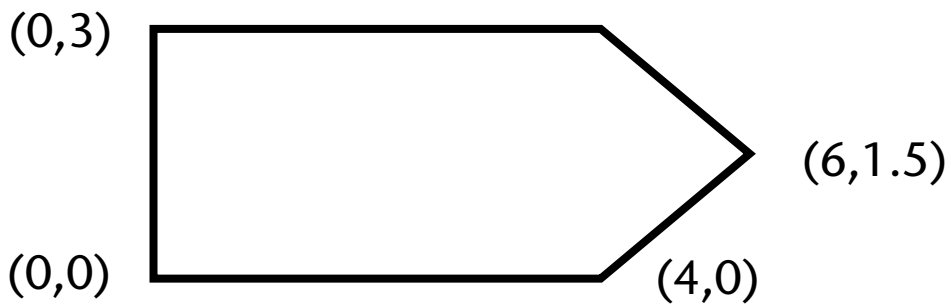
```
glBegin( GL_LINES );
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 1.0, 0.0, 0.0 );
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 0.0, 1.0, 0.0 );
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 0.0, 0.0, 1.0 );
glEnd();
```



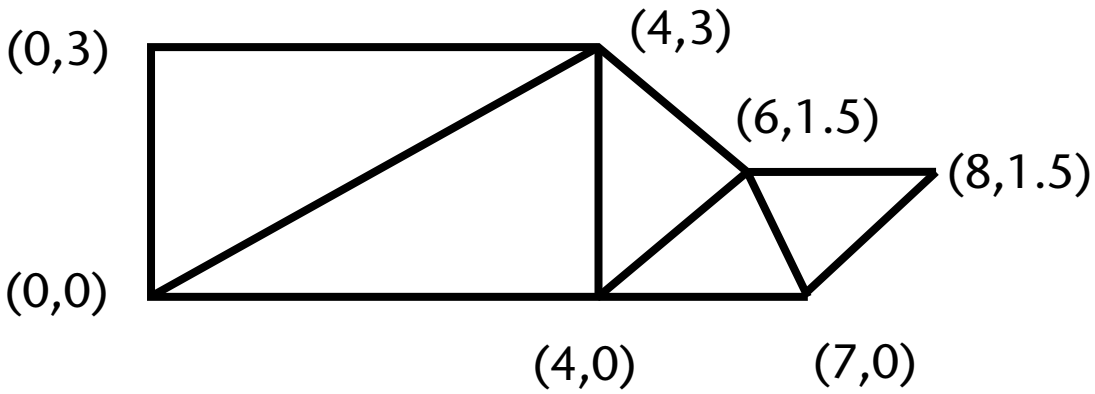
```
glBegin( GL_LINES );
   glColor3f (1.0, 0.0, 0.0);
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 1.0, 0.0, 0.0 );

   glColor3f (0.0, 1.0, 0.0);
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 0.0, 1.0, 0.0 );

   glColor3f (0.0, 0.0, 1.0);
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 0.0, 0.0, 1.0 );
glEnd();
```



```
glBegin(GL_POLYGON);
    glVertex2f( 4.0, 0.0 );
    glVertex2f( 6.0, 1.5 );
    glVertex2f( 4.0, 3.0 );
    glVertex2f( 0.0, 3.0 );
    glVertex2f( 0.0, 0.0 );
glEnd() ;
```



```
glBegin( GL_TRIANGLE_STRIP );
    glVertex2f( 0.0, 3.0 );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 4.0, 3.0 );
    glVertex2f( 4.0, 0.0 );
    glVertex2f( 6.0, 1.5 );
    glVertex2f( 7.0, 0.0 );
    glVertex2f( 8.0, 1.5 );
glEnd() ;
```



- Abschließen der Graphikdarstellung:
  - `void glFlush(void);`

    Diese Funktion sorgt dafür, daß alle OpenGL-Befehle aus dem Command Buffer an die Hardware geschickt werden (ist für die Signalisierung des Endes eines Frames gedacht)

  - `void glFinish(void);`

    Wie glFlush(), wartet aber, bis alle Aufrufe im Framebuffer angekommen sind
- Bei der Verwendung von Qt braucht / sollte man diese Funktionen i.A. nicht selbst aufrufen!

Immediate oder Retained Mode

- Immediate Mode ("direkter Modus"):
  - Primitive werden sofort, wenn sie festgelegt sind, an das Display geschickt (Standard)
  - Graphiksystem hält keine Primitive im Speicher
- Retained Mode ("zurückhaltender Modus"):
  - Primitive kommen in eine sog. Display-Liste (oder *vertex array*, oder ...)
  - Display-Liste kann auf dem Server (Graphikkarte) gehalten werden
  - Kann mehrmals mit unterschiedlichen Eigenschaften gezeichnet werden
  - Performanz wird so erhöht

Zustände

- OpenGL ist ein großes zustandsbasiertes System
- Zustände:
  - Beleuchtung
  - Shading
  - Texture Mapping
  - Wireframe / Solid, mit/ohne Verdeckungstest, mit/ohne Nebel, etc. ...
- Zustände können ein- und ausgeschaltet werden durch **`glEnable`** and **`glDisable`**
- Alle Zustände & Eigenschaften, die geändert werden können, können mit **`glGet`** abgefragt werden

Matrix-Operationen

- OpenGL verwendet 4 Matrizen
  - **`GL_MODELVIEW`**
    - Enthält im Wesentlichen die Transformationen der Objekte
  - **`GL_PROJECTION`**
    - Enthält eine Matrix für die Projektion
  - **`GL_TEXTURE`**
    - Wird zur Bearbeitung von Texturen benötigt (Dehnung, Bewegung, Rotation, etc.)
  - **`GL_COLOR`**
    - Wird für Konvertierung der Farben benötigt

Operationen mit Matrizen

- Definition des aktuell zu bearbeitenden Matrix-Stacks:

```
glMatrixMode( matMode );
// matMode ∈ {GL_MODELVIEW, GL_PROJECTION}
```

- Weitere Matrix- und Stack-Operationen:

```
glLoadIdentity( );
glPushMatrix( );
glPopMatrix( );
glLoadMatrix{fd}( const TYPE *m );
glMultMatrix{fd}( const TYPE *m );
```

- Die oberste Matrix auf dem Stack bestimmt die aktuelle Transformation

Transformationen

- Matrix-Stack
  - **`glPushMatrix, glPopMatrix, glLoad, glMultMatrixf`**
  - Gut bei hierarchisch definierten Figuren, Platzierung
- Transformierung
  - **`glTranslatef(x,y,z) ; glRotatef(θ,x,y,z) ; glScalef(x,y,z)`**
  - Multipliziert an die bestehende Matrix von rechts (letzte wird zuerst angewandt)
- Ebenso **`gluLookAt, gluPerspective`**
  - Erinnerung: **`gluLookAt`** ist eine Matrix wie alle anderen Transformationen auch, hat Einfluss auf Modellansicht
  - Muss im Code vorher vorkommen um Einfluss auf andere Transformationen zu haben
  - Warum gibt es für gewöhnlich keine Ausgabe bei **`gluPerspective`** ?

Modell-Transformationen

- Bewegen eines Objektes

```
glTranslate{fd}( x, y, z );
```

- Rotation eines Objektes um eine beliebige Achse

```
glRotate{fd}( angle, x, y, z );
```

  - Winkel wird in Grad angegeben

- Skalieren eines Objektes

```
glScale{fd}( x, y, z );
```

- Die Transformation wird auf die aktuelle Transformationsmatrix draufmultipliziert und zwar von rechts!

Qt

- Ein API zur Entwicklung von Applikationen mit GUI
- Vor der Entwicklung von Qt:
  - zunächst starr auf die jeweilige Plattform festgelegte Software
    - Beispiele: Windows : GDI / MFC; UNIX : X-Windows / Motif
  - Nachteile: plattformgebunden, meist mühsam (da direkte Programmierung von Nöten)

Konzept einer Qt-Applikation

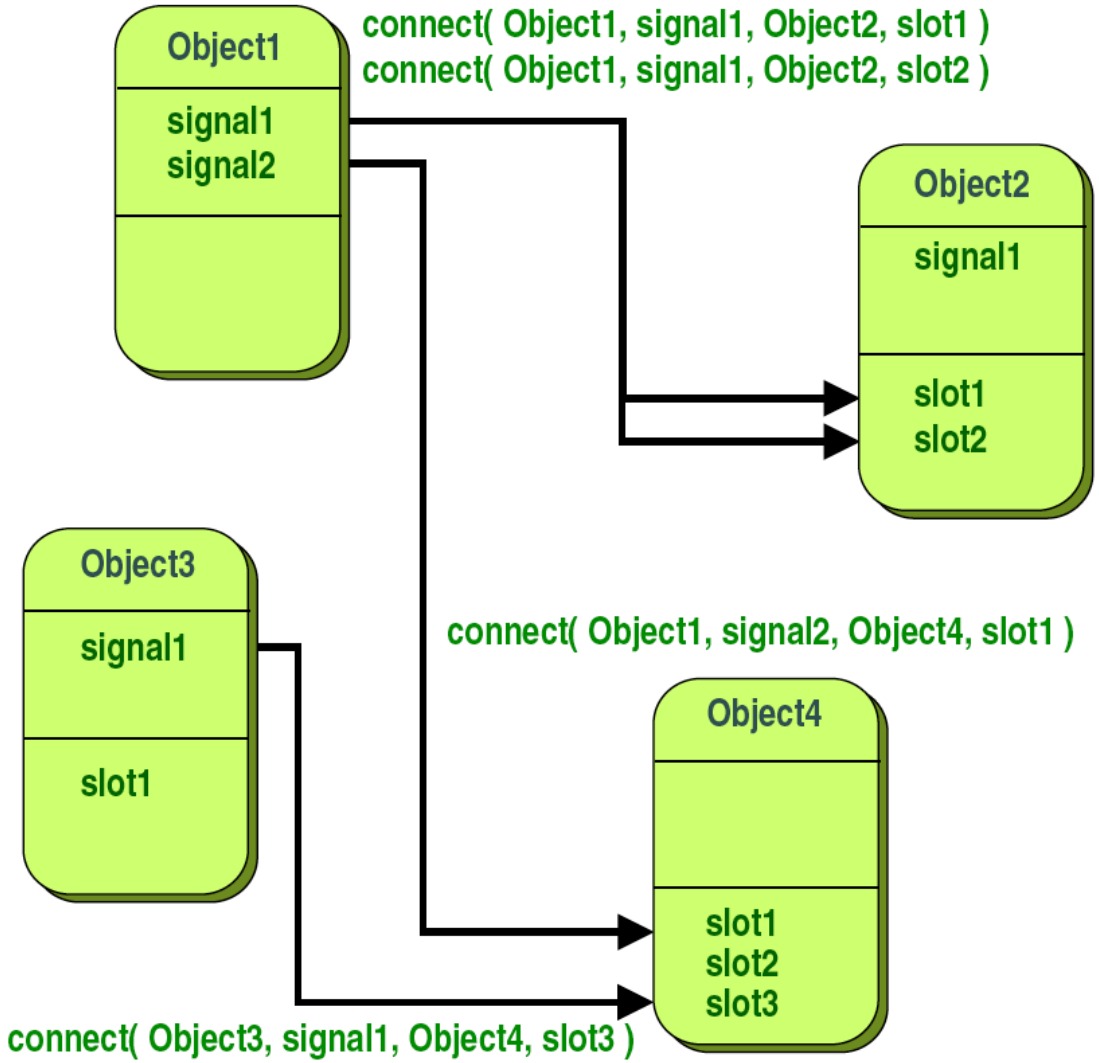
- Merkregel: „Almost everything is a widget !“
  - Widget steht für "window gadget" (= "Fenster-Ding")
  - Widgets sind die Bausteine eines Windows oder GUIs
    - Z.B.: Buttons, Sliders, Graphik-Fenster, Rahmen-Fenster, ...
  - Jede (!) Qt-Applikation enthält Instanzen von Widgets oder speziellen Versionen von Widgets

Ein einfaches QT-Programm

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char* argv[])
{
        QApplication app(argc, argv);
        QPushButton hello("Hallo Welt!");
        hello.resize(100,30);
        hello.show();
        return app.exec();
}
```

Signals und Slots

- liefern Inter-Objekt Kommunikation.
- Idee:
  - Objekte, die nichts voneinander „wissen“, können miteinander verbunden werden
- Jede von QObject abgeleitete Klasse kann **Signal**s deklarieren, die von Funktionen der Klasse ausgestoßen (**emit**) werden können.
- Jede von QObject abgeleitete Klasse kann **Slot**s definieren. Slots sind identisch zu Funktionen; zusätzlich können sie mit Signals "verbunden" werden.
- Zusätzliche Voraussetzung:
  - in der Klassendeklaration muss das Q_OBJECT-Makro aufgerufen werden.
- Die Signals und Slots von Objektinstanzen können miteinander verbunden werden.
  - Wird ein Signal S von Objekt A mit einem Slot T von Objekt B verbunden, und stößt Objekt A das Signal S aus, so wird Slot T von Objekt B aufgerufen. (Erinnerung: Slots sind Funktionen)



Beispiel: connect(button, SIGNAL(clicked()), qApp, SLOT(quit()));

Beispiel: Signals and Slots

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char* argv[])
{
        QApplication app(argc, argv);
        QPushButton hello("Hello World!");
        hello.resize(100,30);
        hello.show();
        QObject::connect( &hello, SIGNAL(clicked()),
                          app,    SLOT(quit())        );
        return app.exec();
}
```

Kompilieren von Qt-Anwendungen

- Jede Plattform hat eigene Tools (*Compiler*, *Linker*, *make*-Programm)
- Qt benötigt zusätzlich Tools (*moc*, *uic*)
- Erstellen von Qt-Applikationen unter Linux:
  - qmake -project generiert Projektdatei (.pro)
  - qmake erstellt aus ihr ein Makefile
  - make erstellt ausführbare Dateien

Qt und OpenGL

- QGLWidget

```
class MyGLDrawer : public QGLWidget
{
Q_OBJECT    // must include this if you use Qt signals/slots

public:
        MyGLDrawer( QWidget *parent, const char *name )
        :  QGLWidget(parent, name) {}

protected:
        void initializeGL()
        {
            // Set up the rendering context, define display
            // lists etc.:
            ...
            glClearColor( 0.0, 0.0, 0.0, 0.0 );
            glEnable(GL_DEPTH_TEST);
            ...
        }
```

```
void resizeGL( int w, int h )
{
    // setup viewport, projection etc.:
    glViewport( 0, 0, (GLint)w, (GLint)h );
    ...
    glFrustum( ... );
    ...
}
void paintGL()
{
    // draw the scene:
    ...
    glRotatef( ... );
    glMaterialfv( ... );
    glBegin( GL_QUADS );
    glVertex3f( ... );
    glVertex3f( ... );
    ...
    glEnd();
 }
```