# Computergraphik I

Prof. Dr.-Ing. Kerstin Müller

## Kapitel 2

## Einführung in OpenGL

# Was ist OpenGL

- OpenGL ist ein Software-Interface für Grafik-Hardware.
- Es besteht aus 150+ verschiedenen Kommandos zur Spezifikation von Objekten und Operationen für interaktive dreidimensionale Anwendungen.
  - Objekte werden aus Primitiven, d.h. Punkte, Linien und Polygonen aufgebaut (keine high-level Funktionalität).
  - Hardware-unabhängig um auf möglichst vielen Architekturen implementiert werden zu können.

# Grundlegende Vorgehensweise (1/2)

- 1. Konstruktion
  - Konstruiere Objekte (Shapes) aus geometrischen Primitiven und damit ein mathematisches Modell.
  - OpenGL kennt als Primitive: Punkte, Linien, Polygone, Bilder und Bitmaps (nicht nur als Bilder, z.B. auch Höhenfeld etc.)
- 2. Anordnung der Objekte in 3D Raum
  - Mathematische Transformationen, d.h. Verschiebung, Rotation, Skalierung etc. um die Objekte im 3D Raum zu platzieren.
  - Auswahl eines Blickwinkels aus den die zusammengesetzte Szene betrachtet wird.

# Grundlegende Vorgehensweise (2/2)

- 3. Berechnung der Farbe der Objekte
  - Die Farbe kann explizit zugewiesen werden
  - Die Farbe kann durch spezielle Beleuchtungs-Bedingungen festgelegt sein.
  - Durch Aufbringen einer sogenannten Textur
  - Oder eine Kombination dieser Vorgehensweisen
- 4. Rasterisierung
  - Umrechnung der mathematischen Darstellung der Objekte und ihrer zugewiesenen Farbe in Pixel auf dem Bildschirm.

# Zusätzliche Operationen und Begriffe

- Während der vier grundlegenden Bearbeitungsschritte können weitere Operationen ausgeführt werden
  - Z.B. die Elimination von Teilen der Objekte, die durch andere Objekte verdeckt werden.
  - Nach der Rasterisierung, aber vor dem Zeichnen auf dem Bildschirm, kann eine Nachbearbeitung der Rasterdarstellung erfolgen (etwa Anti-Aliasing um Treppeneffekte zu vermindern).
- Begriffe:
  - Rendering: Der Prozess bei dem der Computer Bilder aus den Modellen (Objekten) erzeugt (Mathematische Darstellung -> Pixel).
  - Modell: Wird aus geometrischen Primitiven konstruiert.
  - Vertex: Ein Punkt im 3D Raum.

# Ergänzende Bibliotheken

- OpenGL Utility Library (GLU)
  - enthält einige Funktionen zur Programmierhilfe bzw. komplexere Aufgaben (Blickpunkt-Transformationen, NURBS etc.).
  - Teil jeder OpenGL Implementierung
- OpenGL Utility Toolkit (GLUT)
  - Bibliothek, die einfache Funktionen des Fenstersystems bietet (z.B. für Fenstermanagement, Menüs).
  - Soll die Komplexität und Unterschiede der verschiedenen Window-Systeme kapseln.
  - Nicht Teil der OpenGL Implementierung, aber de facto Standard.

# Ein erstes OpenGL Programm

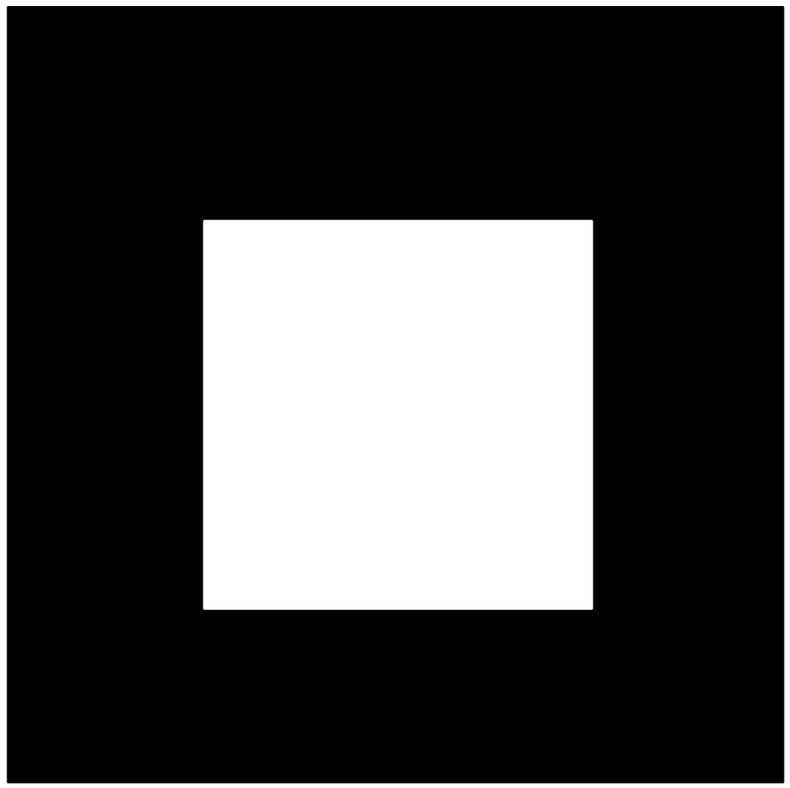
- Zeichnen eines weissen Vierecks auf schwarzem Grund

```
#include <whateverYouNeed.h>

main()
{
OpenAWindow();

glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
glBegin(GL_POLYGON);
glVertex3f(0.25 , 0.25 , 0.0);
glVertex3f(0.75 , 0.25 , 0.0);
glVertex3f(0.75 , 0.75 , 0.0);
glVertex3f(0.25 , 0.75 , 0.0);
glEnd();
glFlush();

UpdateTheWindowAndCheckForEvents();
}
```



# Erläuterung der Kommandos (1/2)

- `OpenAWindow()`:
  - Platzhalter für systemspezifische Routinen zur Initialisierung eines Fensters auf dem Bildschirm.
- `glClearColor(...)` und `glClear(...)`
  - glClearColor(...) legt fest auf welche Farbe der Hintergrund gesetzt werden soll und glClear(...) führt dies tatsächlich durch.
- `glColor3f(...)`
  - Legt die Farbe zum Zeichnen der Objekte fest.
- `glOrtho(...)`
  - Spezifiziert das Koordinatensystem und welche Projektion von 3D nach 2D benutzt werden soll.

# Erläuterung der Kommandos (2/2)

- `glBegin(GL_POLYGON) ... glEnd()`
  - Durch diese Befehle eingefasst wird ein Polygon (Vieleck) durch vier Eckpunkte (Vertices) definiert.
  - Die Argumente sind die x-,y- und z-Koordinate.
    - Das Polygon ist also ein Quadrat in der z=0 Ebene.
- `glFlush()`
  - Stellt sicher, dass die Kommandos nicht nur in einem Puffer gesammelt, sondern auch tatsächlich ausgeführt werden.
  - Z.B. für Transformationen ist das Sammeln sehr viel effizienter als das direkte ausführen (Erklärung folgt in Kapitel 3)

# Datentypen in OpenGL

- Die an Kommandos angehängten Buchstaben deuten die verwendeten Datentypen an (z.B. glColor3f)
- Um Portierbarkeit zu gewährleisten, benutzt OpenGL eigene Datentypen

| Suffix | Datentyp | Äquivalent in C/C++ | OpenGL Typ Definition |
|---|---|---|---|
| b | 8-bit integer | signed char | GLbyte |
| s | 16-bit integer | short | GLshort |
| i | 32-bit integer | int oder long | GLint, GLsizei |
| f | 32-bit floating-point | float | GLfloat |
| d | 64-bit floating-point | double | GLdouble |
| ub | 8-bit unsigned integer | unsigned char | GLubyte, GLboolean |
| us | 16-bit unsigned integer | unsigned short | GLushort |
| ui | 32-bit unsigned integer | unsigned int | GLuint, GLenum, GLbitfield |

# OpenGL als Statusmaschine

- OpenGL kann in verschiedene Zustände (Modes) versetzt werden
  - Jeder Zustand bleibt solange aktiv bis er geändert wird
  - z.B. wird jedes Objekt mit der aktuellen Farbe gezeichnet, solange bis diese geändert wird.
- Weitere Zustände: Textur, Lichtquellen, Linienstärke, Kameraposition, Projektionsart, Material der Objekte etc.
- Viele Zustände werden mit glEnable(...) und glDisable(...) an- und ausgeschaltet.
- Abfrage der zugehörigen Zustandsvariable mit `glIsEnabled(...)` oder z.B. `glGetIntegerv()`

# GLUT Library

- Stellt zunächst standardisierte Window-Funktionalität zur Verfügung
  - **`glutInit(int *argc, char **argv)`**: initialisiert GLUT und verarbeitet Kommandozeilen-Argumente.
  - **`GlutInitDisplayMode(unsigned int mode)`**: Legt Farbmodell fest, single- oder double-buffer, etc.
  - **`glutInitWindowPosition(int x, int y)`**: Bildschirm-Position der oberen linken Ecke des Fensters.
  - **`glutInitWindowSize(int width, int height)`**: Spezifiziert die Größe des Fensters in Pixeln.
  - **`int glutCreateWindow(char *string)`**: Erzeugt ein Fenster mit OpenGL Kontext (return: eindeutiger Identifier für das Fenster).

# OpenGL Programm mit GLUT

- Grundstruktur mit GLUT

```
int main(int argc , char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (250, 250);
glutInitWindowPosition(100, 100);
glutCreateWindow("MyFirstWindow");
init();
glutDisplayFunc(display);
glutMainLoop();
return 0;
}
```

- Noch zu definieren:
  - „display“: Führt die Befehle aus die wiederholt für das Zeichnen jedes Frames notwendig sind.
  - „init“: Enthält Befehle die einmalig zu Beginn ausgeführt werden.

# Init und Display Routinen

- init:

```
void init (void)
{
glClearColor  (0.0, 0.0, 0.0, 0.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho (0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}
```

- display:

```
void display (void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,  1.0, 1.0);
   glBegin(GL_POLYGON) ;
   glVertex3f(0.25, 0.25, 0.0);
   glVertex3f(0.75, 0.25, 0.0);
   glVertex3f(0.75, 0.75, 0.0);
   glVertex3f(0.25, 0.75, 0.0);
   glEnd();
   glFlush();
}
```

# Behandlung von Input-Events

- Registrierung von Callback Kommandos, die bei gewissen Events aufgerufen werden:
  - **`glutReshapeFunc(void(*func)(int w, int h))`**: Die verlinkte Funktion wird aufgerufen wenn sich die Fenstergröße ändert.
  - **`glutKeyboardFunc(void(*func)(unsigned char key, int x, int y))`** Tastatureingaben, d.h. einzelne Buchstaben können mit einer Callback Routine verknüpft werden.
  - **`glutMouseFunc(void(*func)(int button, int state, int x, int y))`** Analog für Maus-Events.
  - **`glutIdleFunc(void(*func)(void))`**: Registriert eine Routine die aufgerufen wird, wenn sonst keine Events abgearbeitet werden müssen. NULL als Argument stoppt die Ausführung der Routine.

# Animationen

- Eine Animation wird durch eine schnelle Folge von Bildern erzeugt, jedes einzelne Bild wird zeilenweise gezeichnet.
- In Pseudocode:

```
open_window();
for(i=0;i<1000000; i++)
{
clear_the_window();
draw_frame(i);
wait_until_a_24th_of_a_second_is_over();
}
```

- Probleme:
  - Man sieht den Prozess des Zeichnens während er passiert.
  - Verschiedene Objekte sind verschieden lang sichtbar.

# Double Buffering für Animationen

- Prinzip Double Buffering:
  - Zwei Bilder werden in je einem Puffer gehalten. Das eine wird angezeigt während das andere gezeichnet wird.
  - Wenn der Zeichenvorgang abgeschlossen ist, werden die Puffer vertauscht.
- Vorteil:
  - Der Betrachter sieht immer nur komplette Bilder.
- In GLUT:
  - void glutSwapBuffers(void);
- Double buffering ist nicht Teil von OpenGL, weil nicht notwendig von jeder Hardware unterstützt.

# Double Buffering für Animationen

- Pseudocode für Double Buffering:

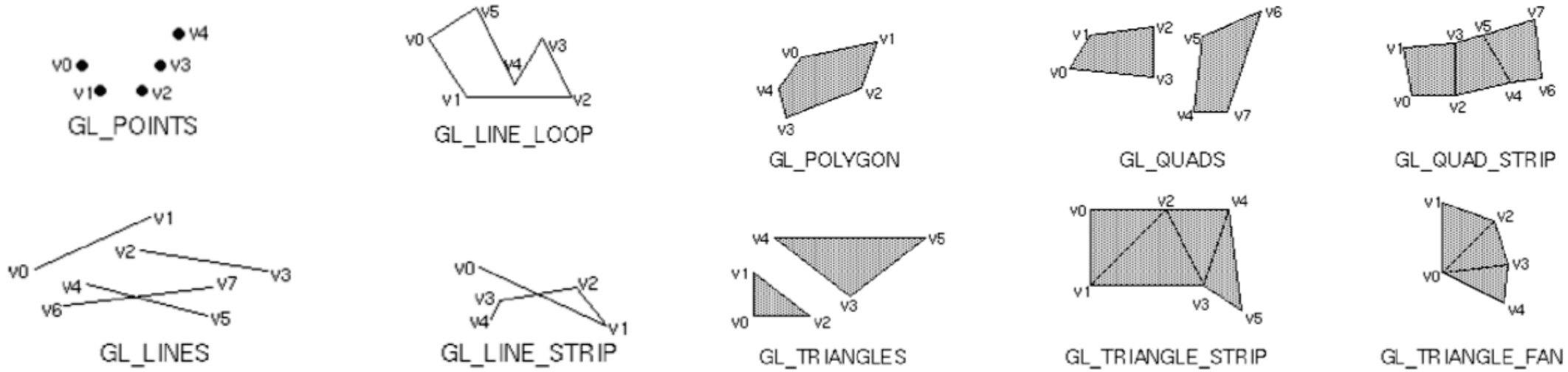
```
open_window_in_double_buffer_mode();
for(i=0;i<1000000; i++)
{
clear_the_window();
draw_frame(i);
swap_the_buffers();
}
```

- Bemerkungen:
  - Um konstante Bild-Wiederholraten zu erreichen warten einige OpenGL Implementierungen bis die Bildschirm-Refresh-Periode vorbei ist, bevor die Puffer vertauscht werden.
  - Bildschirm 60Hz ➡ maximal möglich 60 Hz, auch wenn ein einzelner Frame schneller gezeichnet werden könnte.

# Definition von Objekten

- `void glVertex{2,3,4}{sifd}[v](TYPE coords)`:
  - beschreibt einen Eckpunkt eines geometrischen Objekts. Der Befehl `glVertex*()` sollte zwischen den Befehlen `glBegin(GLenum mode)` und `glEnd()` ausgeführt werden.

- Varianten für `GLenum mode`:



# Objekt-Transformationen

- `void glTranslate{fd}(TYPE x, TYPE y, TYPE z);`
- `void glRotate{fd}(TYPE angle , TYPE x, TYPE y, TYPE z);`
- `void glScale{fd}(TYPE x, TYPE y, TYPE z);`
- Beispiel

```
glMatrixMode (GL MODELVIEW) ;
glLoadIdentity ();
glTranslatef ( -20.0 ,0.0 ,0.0);      /*   translate */
glRotatef (90.0 ,0.0 ,0.0 ,1.0);      /* rotate */
glScalef (1.5 ,0.5 ,1.0);              /*  scale */
glBegin(GL Polygon);
...
```

# Transformationen und Projektionen

- Modelle müssen im 3D-Raum platziert bzw. bewegt werden (Transformationen) und die 3D-Szene in die 2D Bildschirmebene projiziert werden.
- Zur Berechnung werden 4x4 Matrizen benutzt (warum und wie genau ➡ Kapitel 4).
- In OpenGL gibt es je einen Modus zur Bearbeitung von
  - Transformationsmatrizen: `glMatrixMode(GL_MODELVIEW)`
  - Projektionsmatrizen: `glMatrixMode(GL_PROJECTION)`
- Viele solche Matrizen werden auf einem Stack gehalten
  - Warum das funktioniert und sinnvoll ist ➡ Kapitel 3.

# Bearbeiten des Matrix Stacks

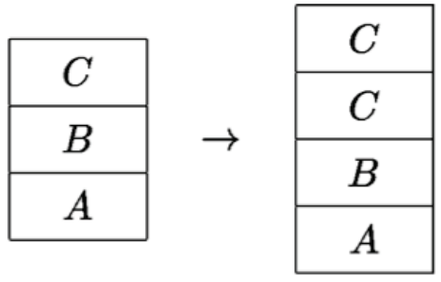
- Ein Matrix Stack ist sinnvoll um hierarchische Modelle zu bearbeiten.
- Beispiel:
  - Ein Auto hat vier gleiche Räder, von denen jedes mit je 5 gleichen Schrauben an der Achse befestigt wird.
  - Es gibt also sinnvollerweise je eine Routine die eine Schraube bzw. ein Rad (ohne Schraube) in sog. lokalen Koordinaten zeichnet.
  - Wenn das Auto gezeichnet wird, soll also die Rad-Zeichen-Routine viermal mit einer jeweils anderen Transformation (Verschiebung) aufgerufen werden. Mit dem Zeichnen jedes Rades soll analog die Schrauben-Zeichen-Routine 5-mal aufgerufen werden.

# Vorgehensweise „umgangssprachlich“

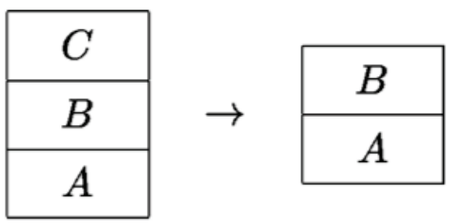
- Nur Räder und Chassis zeichnen
  - 1. Zeichne das Chassis
  - 2. Merke die momentane Position (Mitte)
  - 3. Verschiebe zum rechten vorderen Rad
  - 4. Zeichne ein Rad
  - 5. Vergesse die letzte Verschiebung und gehe zurück (zur Mitte)
  - 6. Merke die momentane Position (Mitte)
  - 7. Verschiebe zum linken vorderen Rad
  - 8. Zeichne ein Rad
  - 9. ...

# Vorgehensweise OpenGL Matrix Stack

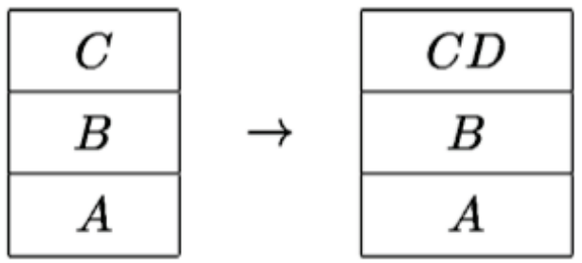
- „Merke“ entspricht dem Kopieren und erneutem Auflegen der obersten Matrix vom Stack
  - `glPushMatrix(void)`




- „Vergesse“ entspricht dem entfernen der obersten Matrix vom Stack
  - `glPopMatrix(void)`




- „`glTranslate`“, „`glRotate`“ etc. entspricht der Multiplikation der obersten Matrix mit einer Matrix D, die die entsprechende Operation ausführt.



# Auto zeichnen OpenGL Pseudocode

- Verwendung von `glPushMatrix()` und `glPopMatrix()`

```
draw_body_and_wheels()
{
   draw_car_body();
   glPushMatrix();
      glTranslatef(40,0,30);
      draw_a_wheel();
   glPopMatrix();
   glPushMatrix();
      glTranslatef(40,0,-30);
      draw_a_wheel();
   glPopMatrix();
   ...
}
```

- 1. Zeichne das Chassis
- 2. Merke die momentane Position (Mitte)
- 3. Verschiebe zum rechten vorderen Rad
- 4. Zeichne ein Rad
- 5. Vergesse die letzte Verschiebung und gehe zurück (zur Mitte)
- 6. Merke die momentane Position (Mitte)
- 7. Verschiebe zum linken vorderen Rad
- 8. Zeichne ein Rad

- Bemerkung:
  - Wenn auch die Schrauben für die Räder gezeichnet werden sollen, muss analog eine weitere Hierarchiestufe in der `draw_a_wheel()` Methode eingefügt werden (➡ Übung).

# Zusammenfassung Modelview Matrix Stack

- Der Modelview Matrix Stack umfasst alle 3D Transformationen des Modells oder Teile davon.
- Der Stack kann üblicherweise bis zu 32 Matrizen enthalten.
- Die Multiplikation der obersten Matrix mit einer Transformationsmatrix wird die neue oberste Matrix.
- Daher kopieren und neu auflegen, falls der vorherige Zustand später wieder hergestellt werden soll.
- Initial ist die oberste Matrix die Einheitsmatrix.

# Projection Matrix Stack

- Alle Projektionsberechnungen werden mit dem Projection Matrix Stack gemacht.
  - Eine Projektion ist in der Regel von 3D nach 2D, daher macht es keinen Sinn mehrere solche Projektionen zu kombinieren.
  - Der Projection Matrix Stack hat in der Regel maximal 2 Elemente.
- Beispiel für den Einsatz einer zweiten Matrix
  - Hilfetext in eine perspektivische 3D-Darstellung einblenden.
  - Positionierung der Schrift schwierig, daher temporär auf parallele Projektion umschalten.
  - Modelview Matrix muss auch passend geändert werden.

```
glMatrixMode(GL_PROJECTION)
glPushMatrix();
    glLoadIdentity();
    glOrtho(..);
    display_some_help_text();
glPopMatrix();
```