C++ Einführung

Dipl.-Medieninform.(FH) Severin S. Todt, MSc.

Fachgruppe Computergraphik und Multimediasysteme

Universität Siegen – Fachbereich 12

Version : 08.11.2004 11:46:00

Übersicht :

7. Exkurs OpenGL – Einführung

6. Funktionen

6.1. Funktionsdeklaration und Funktionsdefinition

- Funktionen ermöglichen eine modulare Strukturierung eines Programms
- Eine Funktion besteht aus einem Block von Anweisungen der ausgeführt wird, wenn er von einer anderen Stelle im Programm aus aufgerufen wird
- Es ist zu unterscheiden zwischen Funktionsdefinition und Funktionsdeklaration

6.1.1. Funktionsdefinition

- Die Funktionsdefinition definiert eine Funktion in Form durch Angabe des Rückgabetyps, des Funktionsbezeichners, der Parameterliste und des Anweisungsblocks samt Vereinbarungen und Anweisungen

<type> function_name (argumentType1 argument1, … argumentTypeN argumentN)
{
[<deklarationen>]
<anweisungen>
}

- Deklarations- und Anweisungsteil stellen den Funktionskörper dar
- <typ> legt den Typ des Rückgabewertes fest
- Die Kombination aus function_name und argumentType1 argument1, … argumentTypeN argumentN) kennzeichnet eindeutig eine Funktion und wird daher als Signatur einer Funktion bezeichnet
  → Zusätzliche Funktionsdefinitionen und –deklarationen mit variierender Parameterliste sind zulässig
  → Weitere Funktionen mit gleicher Parameterliste aber unterschiedlichem Rückgabewert sind nicht erlaubt

```
#include <iostream>

double rechne(double a, double b){
  return a+b;
}

double rechne(int a, int b){
  return a*b;
}

int main(int argc, char** argv){
 cout << "3 + 3 = " << rechne(3.0,3.0) << endl;
 cout << "3 * 3 = " << rechne(3,3) << endl;
}
```

→ 3 + 3 = 6
  3 * 3 = 9

6.1.2. Funktionsdeklaration

- Im Unterschied zur Funktionsdefinition umfasst die Deklaration nur den Rückgabetyp sowie die Signatur der Funktion
- Die Funktionsdeklaration kommt in jedem Source File zu Beginn zum Einsatz, in dem die Funktion aufgerufen wird
- Meistens sind die Funktionsdeklarationen in einem Header zusammengefasst, der eingebunden werden kann (include)

<type> function_name (argumentType1 argument1, … argumentTypeN argumentN);

rechne.cpp:

```
double rechne(double a, double b){
  return a+b;
}

double rechne(int a, int b){
  return a*b;
}
```

rechne.hpp:

```
double rechne(double a, double b);
double rechne(int a, int b);
```

beispiel.cpp:

```
#include <stdio.h>
#include "rechne.hpp"

int main(int argc, char** argv){
  cout << "3 + 3 = " << rechne(3.0,3.0) << endl;
  cout << "3 * 3 = " << rechne(3,3) << endl;
}
```

Compileraufruf : g++ beispiel.cpp rechne.cpp

6.1.3. `void` als Rückgabetyp

- Für den Fall, dass eine Funktion keinen Wert zurückliefern soll oder aber eine Funktion keine Parameter akzeptieren soll kann anstelle der Typangaben das Schlüsselwort **`void`** verwendet werden

```
void printHelp(void) {
  cout << "Ausgabe der Hilfe" << endl;
  return;
}
```

6.2. Parameterübergabe

- Bei der Parameterübergabe kann zwischen Eingabe- und Ausgabeparametern unterschieden werden
- Parameter können als Referenzparameter oder Werteparameter definiert werden
- Parameter können über das Schlüsselwort **`const`** als Eingabeparameter gekennzeichnet und vor Manipulation geschützt werden

6.2.1. Werteparameter

- Werteparameter werden unter Angabe des Typs und des Parameternamen definiert
- Beim Aufruf einer mit Werteparametern definierten Funktion wird eine Kopie der übergebenen Daten als Variable mit funktionsweiter Gültigkeit angelegt
- Die Art des Funktionsaufrufes wird dann als „Call by Value“ bezeichnet

```
void printNumber(double number){
  cout << number << endl;
}
```

Aufruf :

```
double temp;
printNumber(temp);
```

6.2.2. Referenzparameter

- Referenzparameter werden als Referenz auf einen Datentyp mit Hilfe des Referenzoperators definiert
- Beim Aufruf einer mit Referenzparametern definierten Funktion wird die Speicheradresse einer Variablen übergeben
- Die Art des Funktionsaufrufes wird dann als „Call by Reference“ bezeichnet

```
void printNumber(double* number) {
  cout << *number << endl;
}
```

Aufruf :

```
double temp;
printNumber(&temp);
```

6.2.3. Adressparameter

- Adressparameter werden als Adresse auf eine Speicherzelle der Größe eines Datentyps mit Hilfe des Adressoperators definiert
- Beim Aufruf einer mit Adressparametern definierten Funktion wird implizit die Adresse einer übergebenen Variable verwendet
- Der Aufruf erfolgt wie bei einem „Call by Value“ und wird als „Call by Address“ bezeichnet

```
void printNumber(double& number){
  cout << number << endl;
}
```

Aufruf :

```
double temp;
printNumber(temp);
```

6.2.4. Konstante Eingabeparameter

- Über das Schlüsselwort **`const`** geschützte Parameter dürfen nicht verändert werden
- **`const`** ist stets mit Bedacht zu verwenden, es schützt stets nur das Element das unmittelbar hinter dem Schlüsselwort steht
- **`const`** vor dem Typ eines Werteparameters schützt den gesamten Werteparameter

```
void printNumber(const double x){
  x++; // illegal !
  return;
}
```

- const vor dem Typ eines Referenzparameters schützt den Wert der in der durch die Referenz assoziierten Speicherzelle, nicht jedoch den Pointer selbst

```
void printNumber(const double* x){
  (*x)++; // illegal !
   x++;     // völlig legal
  return;
}
```

- const vor dem Referenzoperator eines Referenzparameters schützt den Pointer gegen Veränderung, nicht jedoch den Wert

```
void printNumber(double const* x){
  (*x)++; // völlig legal !
   x++;    // illegal
  return;
}
```

- const vor dem Typ eines Adressparameters schützt die Adresse gegen Manipulation

```
void printNumber(const double& x){
   x++;      // illegal
  return;
}
```

6.3. Funktionspointer

- Um mehr Flexibilität zu gewinnen, ist es möglich, eine Variable als Pointer auf eine Funktion mit einer durch den Rückgabetyp und die Parameterliste festgelegten Struktur zu definieren
- Zur Laufzeit kann die Variablen als Wert eine Funktion annehmen, die dann über die Referenzierung der Variable aufgerufen werden kann
- Die Deklaration eines Funktionspointers erfolgt simultan zur Deklaration einer Funktion, mit der Ausnahme, dass statt des Funktionsnamens in Klammern der Referenzoperator mit dem zu deklarierenden Variablennamen steht und die Parameternamen nicht erscheinen

  <type>(functionPoint_name) (argumentType1, … argumentTypeN);*

- Einer als Funktionspointer deklarierten Variable kann über die Zuweisung des Funktionsnamens einer implementierten Funktion der gleichen Deklaration zugewiesen werden

```
#include <iostream>

double addiere(double a, double b){
  return a+b;
}

double subtrahiere(double a, double b){
  return a+b;
}

int main(int argc, char** argv){
  double (*rechne)(double,double);
  rechne=addiere;
  cout << "3 + 3 = " << rechne(3,3) << endl;
  rechne=subtrahiere;
  cout << "3 - 3 = " << rechne(3,3) << endl;
  return 0;
}
```

6.4. Bibliotheken erstellen und nutzen

- Zur Steigerung der Wiederverwendbarkeit und in großen Projekten ist es sinnvoll, einzelne Problemaspekte in eigenen Paketen, sogenannten Bibliotheken zu bündeln
- Die so entstehenden Module erlauben den Einsatz in späteren Entwicklungsaufgaben, ohne dass der zur Implementierung genutzte Sourcecode zur Verfügung gestellt werden muß
- Die Funktions- und Typdeklaration die in der Bibliothek Verwendung finden werden in einem Headerfile zusammengefasst, dass dann die Nutzung ermöglicht
- Das aus einem neu erstellten Sourcecode erzeugt Objektfile wird im finalen Schritt mit der Bibliothek gelinkt, denn nur hier ist die Implementierung der genutzten Funktionen zu finden

rechne.cpp:

```
double rechne(double a, double b){
  return a+b;
}

double rechne(int a, int b){
  return a*b;
}
```

→ g++ -c rechne.cpp (Objektfile rechne.o wird erzeugt)
→ ar cr librechne.a rechne.o (Statische Bibliothek librechne.a wird ggf. erzeugt und der Bib. rechno.o hinzugefügt)

rechne.hpp:

```
double rechne(double a, double b);
double rechne(int a, int b);
```

beispiel.cpp:

```
#include <iostream>
#include "rechne.hpp"

int main(int argc, char** argv){
  cout << "3 + 3 = " << rechne(3.0,3.0) << endl;
  cout << "3 * 3 = " << rechne(3,3) << endl;
}
```

→ g++ beispiel.cpp –L. –lrechne (beispiel.cpp wird unter Zuhilfenahme der Bibliothek librechne.a (Kurzform –lrechne) übersetzt, die sich im aktuellen Verzeichnis (.) befindet)

7. Exkurs OpenGL – Einführung

7.1. OpenGL Kommando Syntax

- OpenGL Kommandos nutzen alle den Prefix „gl“ gefolgt von einer Reihe zusammengesetzter Wörter, von denen jedes Wort mit einem Großbuchstaben beginnt
- OpenGL Konstanten sind komplett in Großbuchstaben deklariert und beginnen mit „GL“ gefolgt von einem Unterstrich
- Datentypdeklaration in OpenGL beginnen mit „GL“
- Manche Kommandos existieren für eine Variation von Parametertypen, in diesem Fall ist die Anzahl der Parameter zusammen mit deren Typ im Suffix für jede mögliche Variation angegeben

```
glColor3f(GLfloat, GLfloat, GLfloat);
glColor4f(GLfloat,GLfloat,GLfloat,GLfloat);
glColor3i(GLint,GLint,GLint);
```

Beachte:
Reine ANSI-C Notation, in der Überladen der Parameter nicht erlaubt ist

- Mögliche OpenGL Suffixe :

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

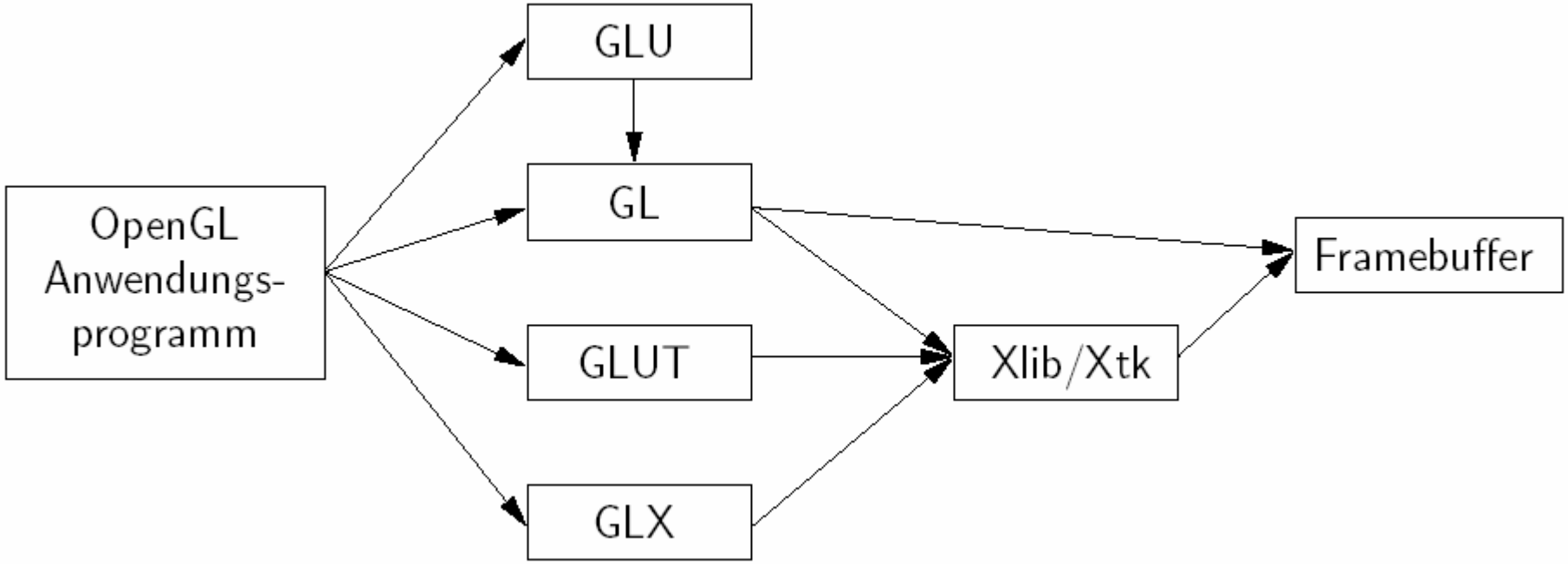
7.2. OpenGL Statusmaschine

- OpenGL implementiert eine Statusmaschine
- Es existieren eine Vielzahl von Statusvariablen, deren Werte zu belegen sind
- Eine Statusvariable behält ihren Wert bei, bis dieser neu gesetzt wird
- Alle Statusvariablen sind bei OpenGL mit Defaultwerten belegt
- Die meisten Statusvariablen bezeichnen Einstellungen die über die Kommandos `glEnable()` oder `glDisable()` aktiviert bzw. deaktiviert werden können
- Der aktuelle Wert aller Statusvariablen kann über glGet*() Funktionen abgerufen werden
  → Je nach Typ der Statusvariable existiert eine spezielle Kommandoimplementierung:

```
glGetBooleanv()
glGetDoublev()
glGetFloatv()
glGetIntegerv()
glGetPointerv()
glIsEnabled()
...
```

7.3. OpenGL verwandte Bibliotheken

- OpenGL bietet einen mächtigen, wenngleich primitiven Befehlssatz zur Graphikprogrammierung auf höherer Ebene
- OpenGL sind auf die Nutzung darunterliegender hardwarenaher Mechanismen angewiesen die einen Zugang zum Window – System bereitstellen
- Diese Mechanismen sind in einer Vielzahl Bibliotheken bereitgestellt



7.3.1. OpenGL Utility Library (GLU)

- GLU stellt Routinen bereit die selbst wieder OpenGL Kommandos nutzen, um
  → Matrix Operationen durchzuführen
  → Berechnungen zur Orientierung durchzuführen
  → Projektionen zu berechnen
  → Polygon Tesselierungen durchzuführen
  → Oberflächen zu rendern
- GLU ist Bestandteil jeder OpenGL Distribution

7.3.2. OpenGL Utility Toolkit (GLUT)

- Für jedes Window System existiert eine Bibliothek die die Funktionalitäten des Window Systems dahingehend erweitert, dass es OpenGL Rendering unterstützt
- Je nach benutztem Betriebssystem und genutztem Window System ist eine andere Bibliothek zu verwenden (glX, xgl, pgl, agl)
- Die zum Rendering in OpenGL Programmen zu verwendenden Kommandos sind abhängig von der genutzten Bibliothek
- GLUT bietet ein Window – System unabhängiges Toolkit, dass die Komplexität variierender Window – System APIs verbirgt
- GLUT stellt weiter einige Funktionen für GL zur Verfügung die das Leben vereinfachen

7.4. Komponenten eines OpenGL Programms

- Zur erfolgreichen Graphikverarbeitung mit OpenGL gilt es, notwendige Komponenten in Form von implementierten Funktionen umzusetzen
- Um die umgesetzten Funktionen nutzen zu können und so eine Anzeige zu erzeugen, sind vorgegebene Schritte zu durchlaufen

7.4.1. Notwendige Funktionskomponenten

- Zur Anzeige ist eine Displayroutine zwingend notwendig, diese ist zu deklarieren und zu implementieren
- Zur Interaktion können Mouse- und Keyboard- Event und andere Eventhandle Routinen notwendig sein, die ebenfalls zu deklarieren und zu implementieren sind
- Zu Animationszwecken sind immer wieder aufgerufene Routinen zur Statusmanipulation nötig

7.4.2. Umsetzungsschritte

1. Initialisierung des Window Systems und erstellen eines Fensters, das zur Anzeige genutzt wird
   Am Beispiel GLUT für Single Buffer Rendering in RGB Mode:
   ```
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow ("Name des Windows");
   glutDisplayFunc(display);
   ```
2. Initialisierung von OpenGL
   ```
   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);
   ```
3. Starten der Prozessverarbeitung des Window Systems
   Am Beispiel GLUT:
   ```
   glutMainLoop();
   ```

7.5. Beispiel Einfaches Rechteck

```
#include <GL/glut.h>

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT); /* clear all pixels  */
   glColor3f (1.0, 1.0, 1.0);
   glBegin(GL_POLYGON);        /* draw white polygon (rectangle) with corners at (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)*/
      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 ();  /* don't wait! start processing buffered OpenGL routines */
}

void init (void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0); /* select clearing color */
   glMatrixMode(GL_PROJECTION); /* initialize viewing values  */
   glLoadIdentity();
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

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

7.5.1. Einführung in make

- Umfangreiche Projekte setzen sich aus einer Vielzahl von Files zusammen
- Kommt es zu Änderungen kann es erforderlich sein, einzelne Module oder das gesamte Projekt neu zu übersetzen
- Bei großen Projekten ist es mühsam, hier den Überblick zu behalten und die Übersetzung händisch zu initiieren
- make hilft bei dieser Aufgabenstellung indem es über Zeitstempel veränderte Files identifiziert und auf Basis eines sogenannten Makefiles notwendige Schritte, wie Übersetzen oder linken initiiert
- Ein Makefile enthält die Regeln, wann, was und wie es auszuführen ist.
  Es besteht aus:
  - → Kommentaren, beginnend mit #
  - → Direktiven, z.B. include zum Einschluß eines weiteren Files
  - → Variablendefinitionen
  - → expliziten Rules
  - → impliziten Rules

- Eine Rule ist dabei wie folgt aufgebaut:
  target: dependencies
  <TAB>command

- Target ist das Was, das zu erstellen ist
- Dependencies legen fest wann das zu geschehen hat, nämlich wenn eines von ihnen jünger ist als target
- Command ist das oder die abzuarbeitenden Kommandos. Ganz wichtig ist der Tabulator am Zeilenanfang!
- Die Verwendung soll an einem ganz einfachen Beispiel mit 2 Rules erläutert werden.

Makefile:

```
hello: hello.cpp
    g++ –L/usr/lib -lglut -lGLU -lGL –lm –o hello hello.cpp
clean:
    -rm -f *.o hello
```

Übersetzen mit : (Erste Regel im Makefile wird angewendet)
→ make

Aufräumen mit : (Regel „clean“ wird direkt aufgerufen)
→ make clean

7.6. Beispiel Einfache Animation

```
#include <GL/glut.h>
#include <stdlib.h>

static GLfloat spin = 0.0;

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glRotatef(spin, 0.0, 0.0, 1.0);
   glColor3f(1.0, 1.0, 1.0);
   glRectf(-25.0, -25.0, 25.0, 25.0);
   glPopMatrix();
   glutSwapBuffers();
}

void spinDisplay(void)
{
   spin = spin + 2.0;
   if (spin > 360.0)
      spin = spin - 360.0;
   glutPostRedisplay();
}

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void reshape(int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

```
void mouse(int button, int state, int x, int y)
{
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(spinDisplay);
         break;
      case GLUT_MIDDLE_BUTTON:
      case GLUT_RIGHT_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(NULL);
         break;
      default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

7.6.1. Vertiefung make

- Die Verwendung von Variablen erleichtern das Leben
- Implizite Regeln erleichert das Leben noch viel mehr
- Verschachtelungen helfen, unter unterschiedlichen Namen gleiche Regeln anzubieten

Makefile:

```
TARGETS = hello double
LIBS = -L/usr/lib -lglut -lGLU -lGL –lm
default: $(TARGETS)
all: default
.cpp.o:
   g++ -c -I/usr/include $<
$(TARGETS): $$@.o
   g++ $@.o $(LIBS) -o $@
clean:
   -rm -f *.o $(TARGETS)
```