Vorbemerkungen

In dieser Dokumentation werden Namen aus den Quelltexten in der Schriftart `Courier` gesetzt. Besonders wichtige Wörter sind *kursiv* hervorgehoben. Der Quelltext selber wurde mit Syntax-Highlighting bearbeitet, um die Lesbarkeit zu erhöhen. Dateinamen sind in der Schriftart Arial kenntlich gemacht.

Um den Umfang meiner Lösungen zu verringern, habe ich große Teile des Quellcodes in Schriftgröße `8` statt `10` gesetzt. Zusätzlich liegt er vollständig in elektronischer Form bei. Trotz aller Vorsichtsmaßnahmen sind gelegentliche ungewollte Zeilenumbrüche im Quellcode möglich.

Als Compiler kommt Visual C++ 6 Professional zum Einsatz, die OpenGL-Bibliothek wird mit der Erweiterung GLUT 3.7 benutzt. Das Betriebssystem ist meist Microsoft Windows, wobei i.d.R. Windows 98 meine Entwicklungsplattform ist. Dem Quellcode liegt immer eine ausführbare Datei bei, die in der sogenannten Release-Einstellung erstellt wurde.

Bei Rückfragen bin ich jederzeit per em@il unter mail@stephan-brumme.com erreichbar.

Aufgabe 1

Ein Großraumbüro wird von den Wänden, der Decke und dem Fußboden eingefasst. In Ausnahmefällen erlauben Fenster einen Blick in Freie, dort ist dann noch zusätzlich die Umgebung zu modellieren, um die Glaubwürdigkeit der Szene zu erhöhen.
Das Büro selbst wird oft durch Stellwände untergliedert, die einzelnen Bürozellen bestehen dann aus Tischen, Stühlen, Computern und diversen Büromaterialien. Eventuell fügt man noch Menschen der Szene hinzu, dann steigt die Datenmenge jedoch erheblich an. Viele der Objekte in diesem Büro sind einander gleich, so dass man unter Umständen z.B. nur einen Stuhl modellieren braucht und sich merkt, wo die Kopien davon entstehen sollen. Damit ist die Szenenmodellierung abgeschlossen.
Selbst wenn man diese Kopiertricks wiederholt anwenden kann, so muss doch mindestens ein Original existieren, dass man zweckmäßigerweise in einem 3D-CAD-Programm erstellt und in die Anwendung einfügt. Zu beachten ist, dass diese Vorlagen dann in einem eigenem Modellraum vorliegen und erst in den Bildraum transformiert werden müssen. Dabei ist es meistens erlaubt, dass die Primitive recht komplexe geometrische Gebilde sind, vielleicht besteht die Sitzfläche der Stühle aus einer gewölbten Freiformfläche.
Selbstverständlich genügt die geometrische Form der Objekte nicht aus, man sollte auch entsprechende Texturen benutzen, um den scheinbaren Detailgrad der Szene zu erhöhen. Ersatzweise reichen auch einfarbige Flächen aus (Stellwände sind oft uni in hellgrau). Diese Attributierung kann auch weitere Oberflächeneigenschaften beinhalten.
Mit der Modellierung der Objekte und ihrer Vervielfältigung sowie Platzierung ist die Anwendungsstufe noch nicht abgeschlossen. Jetzt muss bereits eine Vorverarbeitung erfolgen, die die Primitive der Geometriestufe erzeugt, in der Regel sind dies Dreiecke. Als Beispiel werden die Freiformflächen der Sitzflächen tesselliert, wobei man die Genauigkeit entsprechend der Zielvorstellung anpasst oder gar mehrere Meshes für Level-of-detail (LOD) bereitstellt. In einem Großraumbüro scheint eine Kollisionserkennung (collision detection) unnötig zu sein, da ich von einer statischen Szene ausgehe. Dagegen macht eine sorgfältige Verdeckungsermittlung (occlusion culling) sinnvoll, da die Stellwände viele Bereiche vor den Blicken verstecken.

Die Geometriestufe erzeugt aus den einzelnen Modellen alle notwendigen Kopien (siehe oben erwähntes Beispiel mit den Stühlen) durch Translationen und Rotationen. Andere Objekte benötigen eventuell auch Skalierungen. Genau genommen ist erst jetzt das Großraumbüro mit all seinen Objekten vollständig erzeugt und zum Rendern bereit. Neben den einzelnen Objekten ist manchmal auch das komplette Büro noch zu transformieren.
Die Strukturen innerhalb der Objekte sind jetzt nicht mehr so einfach zu erkennen, da alles bereits als separierte Primitive vorliegt. Gab es vorher Zusammenhänge zwischen den einzelnen Beinen eines Stuhles, so ist jetzt alles in geometrische Objekte mit ihren jeweiligen Attributen zerlegt worden. Der Bildraum ist nun komplett generiert worden.
Die Beleuchtung ist enorm wichtig für die Erfassung der Szene durch den Menschen. Durch die verschiedenen Lichtmodelle (diffus, ambient und Spotlicht) entstehen erst Farbverläufe, die die räumliche Tiefe verdeutlichen. Unterstützt wird dies durch Schattenwürfe und allgemeinen Shading-Verfahren (Radiosity, ...). Gerade

Schattentechniken können Transformationen der Szene erfordern bzw. berechnen zusätzlich Lichtkegel, die sich an den Objekten der Szene schneiden und brechen.
Danach ist eine Projektion erforderlich, die vom Anwendungsgebiet abhängt, für einen technischen Entwurf ist eine orthogonale zu bevorzugen, wesentlich häufiger dürfte man aber eine perspektivische Projektion antreffen. Weil sich die Kamera nicht immer im Ursprung des Modellraums befindet bzw. Verzerrungen wünschenswert sind, müssen erneut alle Primitive transformiert werden. Aus der dreidimensional konstruierten Szene hat man nun eine 2D-Ansicht gewonnen.
Da man i.d.R. keinen Rund-um-Blick erzeugen will, kann man nicht sichtbare Objekte aus der Szene entfernen und teilweise sichtbare entsprechend zurechtschneiden (Clipping).

Die Rasterisierungsstufe setzt die immer noch als (2D-) Vektoren vorliegenden Objekte in diskrete Bildelemente, in Pixel, um. Dieses Zielraster ist oft der Bildschirm, kann aber auch eine sehr große Fläche, wie ein A0-Druckbild sein. Nachdem die Vektoren fragmentisiert wurden, können sie als Linien, Kreise, Ellipsen, Dreiecke oder allgemeine Polygone aufgefasst werden, die wesentlich einfacher zu handhaben sind und für die hochspezialisierte Algorithmen existieren.
Anschließend erfolgt eine Anpassung der Fragmente durch die in der Anwendungsstufe festgelegten Attribute, dies schließt sowohl Texturen, als auch Beleuchtungseffekte mit ein. Diese Arbeit kann bereits stark parallelisiert werden. Nur die sichtbaren Fragmente speichert man danach in den Framebuffer ab. Dabei werden verschiedene Tests, z.B. Z-Tests durchgeführt. Genauso sind auch Pixeloperationen denkbar, wie Gamma-Korrekturen.

Schließlich liest der Videocontroller den Framebuffer aus, um daraus für den Monitor ein Bild zu erzeugen. In älteren Systemen hat man hier Farbkonvertierungen mittels Paletten vorgenommen.

Es ist im allgemeinen nicht einfach, die verschiedenen Transformationen voneinander sauber zu trennen, da sie sowieso in einer Matrix zusammengefasst werden, um den notwendigen Rechenaufwand möglichst stark zu reduzieren. Weiterhin vermischen moderne Grafikkartentreiber, die OpenGL implementieren, die einzelnen Stufen der Pipeline, etwa T&L in Hardware bei der GeForce-Familie. Ebenso sind einige Spezialeffekte, wie Spiegelungen, nur mit mehrerer Renderläufen zu erreichen. Somit kann man meine Erklärungen eher als idealisierte Vorgehensweise betrachten.

Aufgabe 2

Mein Programm reagiert auf Mausklicks (die Taste ist egal), indem es im aktuellen Framebuffer an der aktuellen Position ein Rechteck mittels `glRecti(x, y, x+5, y+5)` zeichnet (siehe `CGMalen::onDraw()`). An diesen Koordinaten bereits vorhandene Rechtecke werden einfach übermalt. Es wird keine Speicherung des Bildes oder der Mausaktionen vorgenommen, lediglich die aktuelle Mausposition befindet sich in den Member-Variablen `m_x` und `m_y`, die zuletzt gedrückte Maustaste speichert `m_mbLastPressed`.

Der Programmrahmen ist vom Lehrstuhl bewusst daraufhin ausgelegt worden, dass möglichst Double-Buffering verwendet wird. Diese im allgemeinen begrüßenswerte Idee erwies sich bei der Aufgabe 2 dann doch als hinderlich. Das Problem besteht darin, dass alle Zeichenoperationen sich im Backbuffer auswirken, für den Benutzer aber immer nur der Frontbuffer sichtbar ist. Zusätzlich benutze ich den bereits vorhandenen Inhalt des Framebuffers und übermale nur, ich zeichne das Bild nicht komplett. Wenn Front- und Backbuffer verschieden sind, so hat man mit Inkonsistenzen zu tun, die sich meist in Flackern einzelner Punkte äußern.
Als Lösung führe ich alle Zeichen- und Löschoperationen zweimal aus, damit beide Buffer stets identisch sind:

```
// needs to be done twice (back- and front buffer)
glRecti(m_x, m_y, m_x+5, m_y+5);
swapBuffers();
glRecti(m_x, m_y, m_x+5, m_y+5);

// Nicht vergessen! Front- und Back-Buffer tauschen:
swapBuffers();
```

Das Standardverhalten der GLUT bezüglich Mausaktionen besteht darin, dass das Niederdrücken einer Maustaste einen Aufruf von `void onButton(MouseButton button, int x, int y)` zur Folge hat. Eine Mausbewegung bei weiterhin gedrückter Taste wirkt sich hingegen als

`void onMove(int x, int y);` aus. Um eine doppelte Implementierung von Code zu vermeiden, speichere ich in `m_mbLastPressed` die zuletzt (oder immer noch) gedrückte Maustaste ab. `onMove` ruft dann einfach unter Benutzung dieser Variable `onButton` auf.

`onButton` selber ist immer noch nicht für Zeichenoperationen zuständig, sondern speichert nur die Mausposition und –taste ab. Erst die letzte Zeile sorgt mit `glutPostRedisplay();` für einen indirekten Aufruf von `onDraw`.

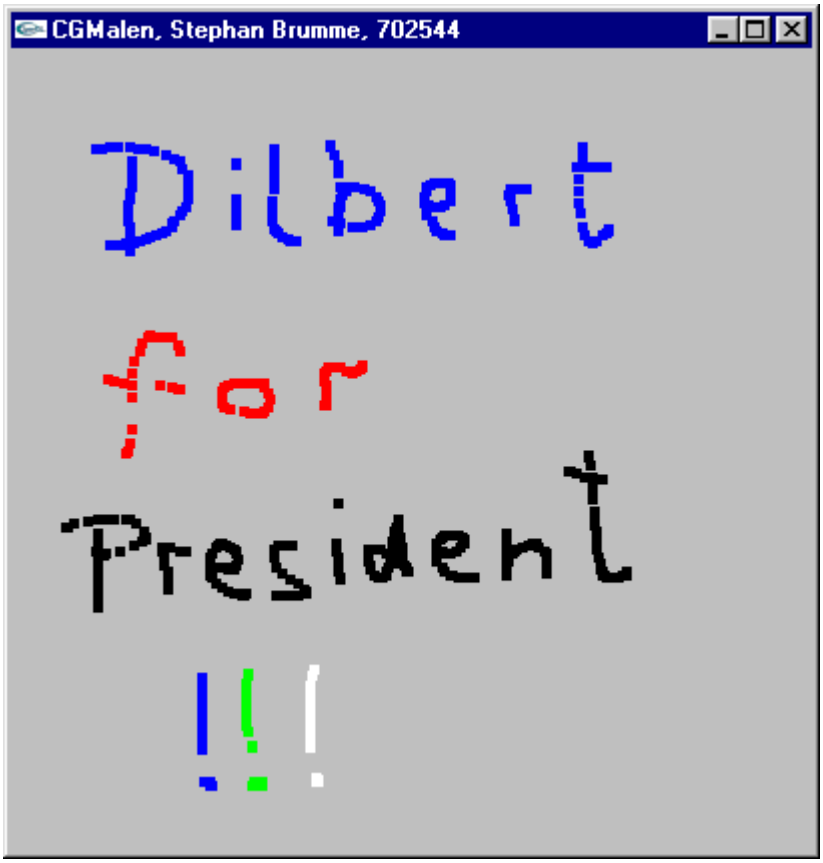
```
void CGMalen::onButton(MouseButton button, int x, int y) {
    //**************************************
    // Speichere die aktuelle Mausposition:
    //**************************************

    // save mouse cursor position
    m_x = x;
    m_y = y;

    // save mouse button for onMove
    m_mbLastPressed = button;

    // force redraw
    glutPostRedisplay();
}
```

Die Startfarbe ist blau auf hellgrauem Hintergrund und kann mit Hilfe der in der Aufgabenstellung geforderten Tasten verändert werden. Aus Gründen einfacherem Quellcodes akzeptiere ich nur Kleinbuchstaben, d.h. *W* bleibt ohne Wirkung, *w* hingegen setzt die aktuelle Zeichenfarbe auf weiss.



Ein noch offenes Problem ist das Ändern der Fenstergröße. Das implizite Löschen von Front- bzw. Backbuffer vernichtet alle bisherigen gezeichneten Figuren. Man könnte diese Situation umgehen, indem man alle Mausaktionen mitprotokolliert, um dann das komplette Bild zeichnen zu können, meine Tests zeigten jedoch, dass sehr schnell 1000 und mehr Objekte zu speichern sind, was zu starkem Speicherverbrauch und einer indiskutablen Performance führt.

Ich habe den Programmrahmen in Form der Klasse `CGApplication` nicht verändert.

Quellcode

source file "cgmalen.h"

```
// Computergraphik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Rahmenprogramm fuer Aufgabenzettel 1


// Stephan Brumme, 702544
// last changes: May 01, 2001

#ifndef CG_SAMPLE_H
#define CG_SAMPLE_H

#include "cgapplication.h"
#include <vector>

class CGMalen : public CGApplication {
public:
    CGMalen();
    virtual ~CGMalen();

    // Ueberschreibe alle diese Ereignisse:
    virtual void onInit();
    virtual void onDraw();
    virtual void onSize(unsigned int newWidth, unsigned int newHeight);
    virtual void onButton(MouseButton button, int x, int y);
    virtual void onMove(int x, int y);
    virtual void onKey(unsigned char key);

private:
    MouseButton m_mbLastPressed;
    GLint       m_x, m_y;
};

#endif // CG_SAMPLE_H
```

source file "cgmalen.cpp"

```
// Computergraphik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Rahmenprogramm fuer Aufgabenzettel 1


// Stephan Brumme, 702544
// last changes: May 01, 2001

#include "cgmalen.h"

CGMalen::CGMalen() : m_x(-10), m_y(-10) {
}

CGMalen::~CGMalen() {
}

void CGMalen::onInit() {
    //*************************************
    // Setze die Hintergrundfarbe auf grau:
    //*************************************

    // gray background with no alpha
    glClearColor(0.75, 0.75, 0.75, 0.0);
```

```
    //**************************************
    // Loesche den Farbspeicher (wird mit der Hintergrundfarbe
    // gefuellt):
    //**************************************

    // needs to be done twice (back- and front buffer)
    glClear(GL_COLOR_BUFFER_BIT);
    swapBuffers();
    glClear(GL_COLOR_BUFFER_BIT);

    //**************************************
    // Setze die Zeichenfarbe
    //**************************************

    // blue is initially set color
    glColor3f(0.0, 0.0, 1.0);
}

void CGMalen::onDraw() {

    //**************************************
    // Zeichne das Rechteck an der gespeicherten Position:
    //**************************************

    // needs to be done twice (back- and front buffer)
    glRecti(m_x, m_y, m_x+5, m_y+5);
    swapBuffers();
    glRecti(m_x, m_y, m_x+5, m_y+5);

    // Nicht vergessen! Front- und Back-Buffer tauschen:
    swapBuffers();
}

void CGMalen::onSize(unsigned int newWidth, unsigned int newHeight) {
    if((newWidth > 0) && (newHeight > 0)) {
        // Passe den OpenGL-Viewport an die neue Fenstergroesse an:
        glViewport(0, 0, newWidth - 1, newHeight - 1);

        // Passe die OpenGL-Projektionsmatrix an die neue
        // Fenstergroesse an:
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, newWidth - 1, 0.0, newHeight - 1);

        // Schalte zurueck auf die Modelview-Matrix und
        // initialisiere sie mit der Einheitsmatrix:
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

//      glutPostRedisplay();
    }
}

void CGMalen::onButton(MouseButton button, int x, int y) {
    //**************************************
    // Speichere die aktuelle Mausposition:
    //**************************************

    // save mouse cursor position
    m_x = x;
    m_y = y;

    // save mouse button for onMove
    m_mbLastPressed = button;

    // force redraw
    glutPostRedisplay();
}

void CGMalen::onMove(int x, int y) {
    // pass current position to onButton
    onButton(m_mbLastPressed, x, y);
}

void CGMalen::onKey(unsigned char key) {
```

```
    //**************************************
    // reagiere auf Tasteneingabe
    //**************************************

    switch (key)
    {
    // change current color
    // red
    case 'r': glColor3f(1.0, 0.0, 0.0);
              break;
    // green
    case 'g': glColor3f(0.0, 1.0, 0.0);
              break;
    // blue
    case 'b': glColor3f(0.0, 0.0, 1.0);
              break;
    // black
    case 's': glColor3f(0.0, 0.0, 0.0);
              break;
    // white
    case 'w': glColor3f(1.0, 1.0, 1.0);
              break;

    // clear view (clear both buffers !!!)
    case 'c': glClear(GL_COLOR_BUFFER_BIT);
              swapBuffers();
              glClear(GL_COLOR_BUFFER_BIT);
              break;

    // quit program
    case 'q': exit(0);
    }
}

// Hauptprogramm
int main(int argc, char* argv[]) {
    // Erzeuge eine Instanz der Beispiel-Anwendung:
    CGMalen sample;

    // Starte die Beispiel-Anwendung:
    sample.start("CGMalen, Stephan Brumme, 702544");
    return(0);
}
```

source file "cgapplication.h"

```
// Computergraphik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Rahmenprogramm fuer Aufgabenzettel 1

#ifndef CG_APPLICATION_H
#define CG_APPLICATION_H

#include <GL/glut.h>

// einige haeufig verwendete Header-Dateien
#include <assert.h>
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

class CGApplication {
public:
    // Die Klasse "CGApplication" stellt ein Rahmenprogramm bereit,
    // mit dem sie ihre Aufgaben loesen koennen. Dazu muessen sie
    // eine Klasse hiervon ableiten, und eine oder mehrere der
    // weiter unten deklarierten on*()-Methoden ueberschreiben.
    // Diese Klasse ist ein sogenanntes "Singleton", d.h.: Es kann
    // immer nur eine einzige Instanz von dieser (oder einer
    // abgeleiteten) Klasse erzeugt werden!
```

```
    CGApplication();
    virtual ~CGApplication();

    // Startet die Anwendung und muss im Hauptprogramm aufgerufen werden.
    void start(const char* windowTitle = "CGApplication",
               bool doubleBuffering = true,
               unsigned long windowWidth = 400, unsigned long windowHeight = 400);

    // Diese Methode wird nur einmal beim Start der Anwendung aufgerufen
    // und dient zur Initialisierung von OpenGL (hier kann z.B. die
    // Hintergrundfarbe fuer das Anwendungsfenster definiert werden). Sie
    // kann von einer abgeleiteten Klasse ueberschrieben werden.
    virtual void onInit();

    // Jedesmal wenn der Fensterinhalt neu gezeichnet werden muss,
    // wird diese Methode aufgerufen. Sie muss von einer abgeleiteten
    // Klasse ueberschrieben werden!
    virtual void onDraw() = 0;

    // Jedesmal wenn sich die Fenstergroesse geaendert hat, wird diese
    // Methode aufgerufen (die Methode onDraw() wird im Anschluss
    // automatisch aufgerufen). Sie muss von einer abgeleiteten Klasse
    // ueberschrieben werden!
    virtual void onSize(unsigned int newWidth, unsigned int newHeight) = 0;

    // Spezifiziert die beiden Konstanten "LeftMouseButton" und
    // "RightMouseButton".
    enum MouseButton { LeftMouseButton, RightMouseButton };

    // Jedesmal wenn eine Maustaste im Fenster gedrueckt wird, wird diese
    // Methode mit der entsprechenden Taste ("LeftMouseButton" oder
    // "RightMouseButton") und den zugehoerigen Koordinaten (x und y)
    // aufgerufen. Die Koordinaten werden als Pixel uebergeben, wobei der
    // Koordinaten-Ursprung die linke untere Fensterecke ist. Diese
    // Methode kann von einer abgeleiteten Klasse ueberschrieben werden.
    virtual void onButton(MouseButton button, int x, int y);

    // Jedesmal wenn die Maus mit gedrueckter Maustaste ueber das Fenster
    // bewegt wird, wird diese Methode mit den entsprechenden Koordinaten
    // aufgerufen (siehe auch Methode onButton()). Diese Methode kann von
    // einer abgeleiteten Klasse ueberschrieben werden.
    virtual void onMove(int x, int y);

    // Jedesmal wenn eine Taste gedrueckt wird, wird diese Methode
    // aufgerufen.
    virtual void onKey(unsigned char key);

    // Diese Methode sollte nach Beendigung aller Zeichenoperationen fuer
    // ein Bild (normaler Weise am Ende der Methode onDraw()) aufgerufen
    // werden, um den Front- mit dem Back-Buffer zu tauschen! Wird dieser
    // Aufruf ausgelassen, so werden die Zeichenoperationen nicht sichtbar!!!
    void swapBuffers();

private:
    // Verbiete den Copy-Konstruktor und den Zuweisungsoperator, da
    // diese Klasse ein Singleton ist und somit nicht kopiert werden
    // darf!
    CGApplication(const CGApplication&);
    CGApplication& operator=(const CGApplication&);
};

#endif // CG_APPLICATION_H
```

source file "cgapplication.cpp"

```
// Computergraphik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Rahmenprogramm fuer Aufgabenzettel 1

#include "cgapplication.h"
```

```
// Diese Variable enthaelt einen Zeiger auf die aktive Anwendung.
static CGApplication* activeApplication_ = 0;

CGApplication::CGApplication() {
    assert(!activeApplication_); // Es ist nur eine aktive Anwendung erlaubt!!!
    activeApplication_ = this;
}

CGApplication::~CGApplication() {
    activeApplication_ = 0;
}

// Auf diese Ereignisse wird default-maessig nicht reagiert:
void CGApplication::onInit() { }
void CGApplication::onButton(MouseButton button, int x, int y) { }
void CGApplication::onMove(int x, int y) { }
void CGApplication::onKey(unsigned char key) { }

// glut-Callback fuer den Zeichenaufruf
static void displayFunc() {
    assert(activeApplication_);
    activeApplication_->onDraw();
}

// glut-Callback fuer die Aenderung der Fenstergroesse
static void reshapeFunc(int newWidth, int newHeight) {
    assert(activeApplication_);
    activeApplication_->onSize(newWidth, newHeight);
}

// glut-Callback fuer das Druecken einer Maustaste
static void mouseFunc(int button, int state, int x, int y) {
    assert(activeApplication_);
    if(state == GLUT_DOWN) {
        y = glutGet(GLenum(GLUT_WINDOW_HEIGHT)) - y; // Koordinaten-Ursprung: links unten!!!
        if(button == GLUT_LEFT_BUTTON) {
            activeApplication_->onButton(CGApplication::LeftMouseButton, x, y);
        } else {
            activeApplication_->onButton(CGApplication::RightMouseButton, x, y);
        }
    }
}

// glut-Callback fuer das Ziehen bei gedrueckter Maustaste
static void motionFunc(int x, int y) {
    assert(activeApplication_);
    y = glutGet(GLenum(GLUT_WINDOW_HEIGHT)) - y; // Koordinaten-Ursprung: links unten!!!
    activeApplication_->onMove(x, y);
}

// glut-Callback fuer einen Tastendruck
static void keyboardFunc(unsigned char key, int x, int y) {
    assert(activeApplication_);
    activeApplication_->onKey(key);
}

void CGApplication::start(const char* windowTitle, bool doubleBuffering,
                          unsigned long width, unsigned long height) {
    // Fordert ein OpenGL-Fenster im RGB-Format mit oder ohne Double-Buffering an:
    glutInitDisplayMode(GLUT_RGB | (doubleBuffering ? GLUT_DOUBLE : GLUT_SINGLE));

    // Legt die Fenstergroesse fest:
    glutInitWindowSize(width, height);

    // Erzeugt das OpenGL-Fenster mit dem entsprechenden Namen:
    glutCreateWindow(windowTitle);

    // Registriere die oben definierten Callbacks fuer die
    // entsprechenden glut-Ereignisse:
    glutDisplayFunc(displayFunc);
    glutReshapeFunc(reshapeFunc);
    glutMouseFunc(mouseFunc);
    glutMotionFunc(motionFunc);
    glutKeyboardFunc(keyboardFunc);
```

```
    // Rufe die Methode zur Initialisierung von OpenGL auf:
    onInit();

    // Starte die Verarbeitung der glut-Ereignisse:
    glutMainLoop();
}

void CGApplication::swapBuffers() {
    glutSwapBuffers();
}
```