Grafische Datenverarbeitung
Sommersemester 2008

Wii Tetris GL

Dozent: Prof. Dr. math. Bunse

Dipl.-Ing. (FH) Christian Benjamin Ries
Christian.Ries@linux-sources.de

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Ich versichere an Eides Statt, dass ich nach bestem Wissen die reine Wahrheit gesagt und nichts verschwiegen habe.

Vor Aufnahme der obigen Versicherung an Eides Statt wurde ich über die Bedeutung der eidesstattlichen Versicherung und die strafrechtlichen Folgen einer unrichtigen oder unvollständigen eidesstattlichen Versicherung belehrt.

_____

(Ort, Datum)

_____

(Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

An der Fachhochschule Bielefeld wird von Prof. Dr. math. Wolfgang Bunse die Lehrveranstaltung „Grafische Datenverarbeitung" angeboten. In dieser Vorlesung werden Grundlagen und das Verständniss für Methoden der interaktiven grafikerzeugenden und grafikverändernden Datenverarbeitung vermittelt.

Für das Bestehen dieser Vorlesung wird ein durchgeführtes Projekt abverlangt. In dieser Arbeit geht es um dieses Projekt.

2 Idee

In den letzten Jahrzehnten existierte eine Fülle von verschiedenen Spielekonsolen. Angefangen im Jahre 1972 als Ralph Baer die erste Spielekonsole entwickelte, gefolgt von zahlreichen Varianten und Arten. Doch erst ab 1985 setzten sich die Konsolen durch und

eroberten den Multimediamarkt. Die bekanntesten Konsolen dieser Zeit sind wohl das NES (Nintendo Entertainment System), das Sega Master System und der Atari 7800.

Die heutigen Spielekonsolen haben nur noch eines mit den damaligen Geräten gemeinsam: den Spieler, der diese bedient. Namen wie Xbox 360 (Microsoft), Playstation 3 (Sony) und Wii (Nintendo) sind heutzutage in aller Munde. Angetrieben durch geballte Rechenpower und Speicherkapazität, berechnen diese Megamaschinen die realistischsten Szenen und Effekte und zaubern wahrliche Kunst auf die heutigen Plasma-Fernseher.

Somit kam ich zu der Idee, die Technologien und Konzepte dieser Spielekonsolen für meine Zwecke zu verwenden! Mit der von Nintendo entwickelten Spielekonsole „Wii" wird ein neuer Weg der Benutzer-Maschinen-Interaktion bestritten. Die Wii Konsole liefert zwei Steuereinheiten mit, zum einem die Wii-Fernbedienung (im folgenden „Wiimote" genannt) und den sogenannten Nunchuk (Abbildung 1).

Abbildung 1: Steuerungselemente der Nintendo Wii (Nunchuck und Wiimote v.L.)

Der Wiimote funktioniert kabellos und reagiert durch einen eingebauten Beschleunigungssensor auf die Bewegungen des Trägers, die in Translationen und Rotationen im Spiel umgesetzt werden. Zusätzlich enthält der Wiimote Steuerungsknöpfe, die für weitere Funktionen verwendet werden. Der Nunchuck wird über ein Kabel an die Wiimote angeschlossen und besitzt auch einen Beschleunigungssensor und zudem ein sehr empfindliches Steuerungskreuz mit zwei weiteren Steuerungsknöpfen.

In dieser Arbeit geht es um ein Modernisieren des Spieleklassikers „Tetris", in Art der Darstellung, sowie der Steuerung. Für die Darstellung wird die plattform- und programmiersprachen-unabhängige API[1] OpenGL und als Steuerung die oben genannten Steuerungselemente verwendet. Auf die Festlegung der Steuerung in diesem Tetris wird im nachfolgenden Abschnitt eingegangen.

3 Tetris, das Spiel

3.1 Spielregeln

Tetris ist eine Art Echtzeitpuzzle. Nacheinander erscheinen Figuren aus vier Blöcken oder einem Block und sinken zum Boden einer Grube. Der Spieler muß diese freihalten, indem er die Figuren bewegt und rotiert. Jedesmal, wenn eine Blockzeile vollständig gefüllt ist,

[1] Application Programming Interface, Eine Programmierschnittstelle, die von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird.

verschwindet sie und alle Blöcke oberhalb fallen herab. Nach jeweils 20 gelöschten Zeilen wird das Spiel ein wenig schneller. Das Spiel ist vorbei, wenn die Grube zu voll für die nächste Figur ist.

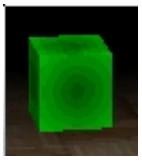

Abbildung 2: Kubus, Größe 1x1x1, 1 Stein

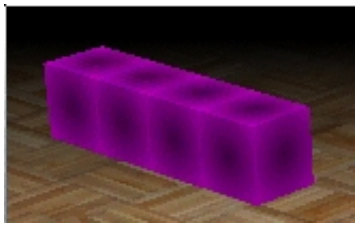

Abbildung 3: Linie, Größe 4x1x1, 4 Steine

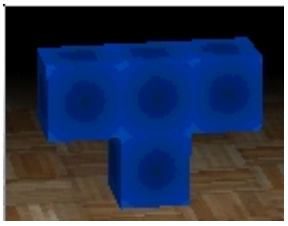

Abbildung 4: Pyramide, Größe 3x2x1, 4 Steine

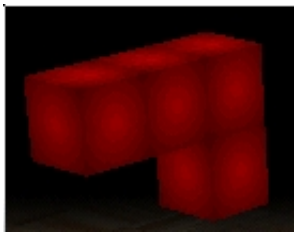

Abbildung 5: Rechtwinkel, Größe 3x2x1, 4 Steine

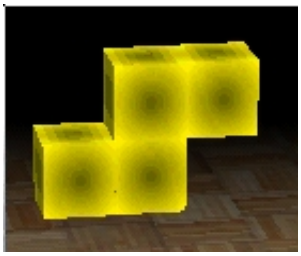

Abbildung 6: Schlange, Größe 3x2x1, 4 Steine

3.2 Punktevergabe

Alle Bausteine haben einen festzugeordneten Punktewert, der beim erfolgreichen Setzen der Gesamtpunktzahl hinzuaddiert wird. Der Kubus bringt 10 Punkte, die Linie 30 Punkte, mit der Pyramide werden 100 Punkte addiert, der Rechtwinkel zählt 50 und die Schlange 90 Punkte.

3.3 Die Spielwiese

In Abbildung 7 wird das Spiel im pausierten Modus abgebildet. Die schachbrettartige Fläche ist der Bereich, auf dem die Bausteine gesetzt werden. Diese Fläche ist 10x10 Einheiten groß. Die roten Linien, zwischen aktuellem Baustein und der Spielfläche, dient als Hilfe zum Setzen der Bausteine.

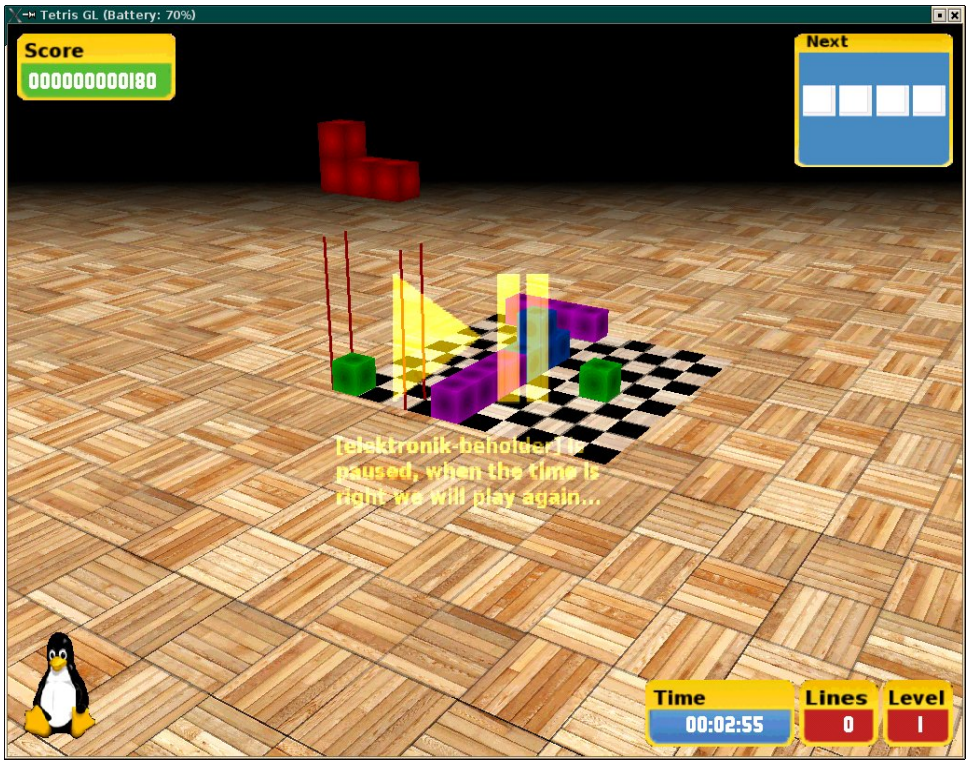

Abbildung 7: Spielansicht von Tetris GL

Oben links werden die erspielten Punkte dargestellt, oben rechts der nächste Baustein angezeigt, der nach dem Setzen des Aktuellen gespielt werden muss. Unten rechts befinden sich statistische Werte, die aktuelle Spielzeit, die erspielten Linien und das erreichte Level. In der Titelleiste wird der aktuelle Batteriestand des Wiimote angezeigt, im Bild sind es *70%*.

3.4 Spielstart

Das Spiel wird normal in einer Konsole gestartet. Zwei Startparameter stehen zur Verfügung, die das Verhalten des Spiels beinflussen. Tabelle 1 listet diese auf. Im Listing 1 wird ein exemplarisches Beispiel zum Starten von Tetris GL aufgezeigt.

```
1 $./tetris --wii --audio
```

Listing 1: Textzeile zum Starten von Tetris GL in einem Konsoleneingabefenster.

Spielparameter beim Programmstart

–wii	Es wird eine Verbindung zum Wiimote hergestellt.
–audio	Die Soundunterstützung wird geladen.

Tabelle 1: Verwendbare Parameter beim Start des Spiels

3.5 Steuerung

Die Hauptsteuerung geschieht mit denen in Kapitel 2 erwähnten Steuerungselementen, Wiimote und Nunchuck. Die Steuerung ist von vornherein auf die in Tabelle 2 definierten Kombinationen festgelegt und bezieht sich zudem auf die Tasten, die in Abbildung 8 und 9 dargestellt sind.

Steuerungsmöglichkeiten mit dem Wiimote und des Nunchuck

Kippung des Wiimote auf der Y-Achse "Yaw" (siehe Abbildung 8)	Der aktuelle Baustein bewegt sich auf der Z-Achse.
Kippung des Wiimote auf der X-Achse "Roll" (siehe Abbildung 8)	Der aktuelle Baustein bewegt sich auf der X-Achse.
Control, Klick auf links/rechts	Drehung des Bausteins auf der Z-Achse im Spiel.
Control, Klick nach oben/unten	Drehung des Bausteins auf der X-Achse in Spiel.
1	Schaltet die Ansicht in den Wireframemodus, alle Bausteine werden als Gittergeflecht angezeigt.
2	Pausiert das Spiel oder startet das Spiel neu, wenn es beendet worden ist.
A	Drehung des Bausteins auf der Y-Achse im Spiel.
B	keine Zuordnung
C	Beschleunigter Fall des aktuellen Bausteins in die Grube.
Z	Drehung des Bausteins auf der Y-Achse im Spiel.
Control, Klick nach oben/unten + gedrückter B Taste	Heraus-/Hineinzoomen des Beobachters.
Control, Klick auf links/rechts + gedrückter B Taste	Beobachtungspunkt des Spielers nach rechts und links verändern.
Stick	Drehung des Beobachterpunktes um die X- und Y-Achse im Spiel.
Home	Beendet das Spiel und Programm.

Tabelle 2: Zuordnung der Steuerungsmöglichkeiten des Wiimote und des Nunchucks

Sollten dem Spieler keine Steuerelemente des Nintendo Wii zur Verfügung stehen, kann als Ersatz die Standardtastatur verwendet werden. Die Tastenkombinationen sind in Tabelle 3 aufgelistet. Diese sind festzugeordnet und können im laufenden Spiel nicht verändert werden.

4 Architektur

Als Basis für die Entwicklung von Tetris GL kommt **SDL**[2], mit zusätzlichen Bibliotheken, zum Einsatz. Diese Bibliotheken sind:

[2]Simple Directmedia Layer, www.libsdl.org

Steuerungsmöglichkeiten mit der Tastatur

Pfeiltaste links/rechts	Drehung des Bausteins um die Z-Achse.
Pfeiltaste oben/unten	Drehung des Bausteins um die X-Achse.
Entfernen	Drehung des Bausteins auf der Y-Achse.
Seite runter	Drehung des Bausteins auf der Y-Achse.
D/A	Translation des Bausteins auf der X-Achse.
Q/E	Translation des Bausteins auf der Z-Achse.
P	Pausiert das Spiel oder startet das Spiel neu, wenn es beendet worden ist.
Enter	Beschleunigter Fall des aktuellen Bausteins in die Grube.
W/S	Heraus-/Hineinzoomen des Beobachters.
R/T/F/G	Drehung des Beobachterpunktes um die X- und Y-Achse im Spiel.
Esc	Beendet das Spiel und Programm.

Tabelle 3: Zuordnung der Steuerungsmöglichkeiten des Wiimote und des Nunchucks

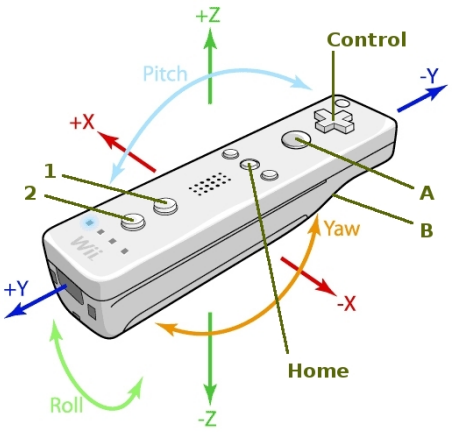

Abbildung 8: Koordinatensystem und Tasten des Wiimote

Abbildung 9: Tasten des Nunchuck.

- SDL_mixer, zur Ausgabe von Sound.
- SDL_ttf, zur Darstellung von TrueType-Fonts.

Zum Laden von Texturen, jeglichen Dateiformates kommt die Grafikbibliothek **ImageMagick**[3] zum Einsatz. Als Unterstützung dieser Funktionalitäten in der Programmiersprache C++ wird **Magick++**[4] genutzt. Dies ist die objektorientierte C++ API[5].

Zur Visualisierung der grafischen Elemente kommt der Industriestandard **OpenGL** zum Zuge.

[3] www.imagemagick.org
[4] www.imagemagick.org/Magick%2B%2B/
[5] Application Programming Interface, Eine Programmierschnittstelle ist eine Schnittstelle, die von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird.

Die Ansteuerung der Wii Steuerungselemente geschieht mit der noch jungen Bibiliothek **CWiid**[6]. Diese Bibilothek bietet Möglichkeiten, die Steuerungselemente über Bluetooth zu steuern und im Spiel damit die Interaktion aufzubauen.

Zur Zeit läuft das Spiel nur unter Linux mit folgender Konfiguration (Testumgebung):

- Grafikkarte: GeForce 7900 GS/PCI/SSE2
- OpenGL Version: 2.1.2 NVIDIA 169.12
- SDL Version: 1.2.12
- SDL TTF Version: 2.0.9
- SDL Mixer Version: 1.2
- CWiid Version: 0.6.0

5 Programmablauf

Abbildung 10 verdeutlicht den grundsätzlichen Ablauf eines Spiels. Beim Start des Programms, werden alle wichtigen Daten und Grundstrukturen initialisiert, u.a. wird die Verbindung zur Wiimote hergestellt, wenn dies beim Programmstart angegeben wurde (siehe dazu Kapitel 3.4). Als Basis für das Programm muss mindestens das SDL Framework erfolgreich geladen sein. SDL übernimmt die Verwaltung aller Ausgaben unter dem X11 Windowmanager. Zudem muss die OpenGL Schnittstelle erfolgreich eingerichtet sein, ohne die keine 3D Darstellungen/Ausgaben stattfinden.

Sollten bis hier keine Fehler aufgetreten sein, springt das Programm in die Hauptschleife. In der Hauptschleife werden alle spielrelevanten Berechnungen durchgeführt. Im Kapitel 7.3 wird auf die einzelenen Komponenten, die für die Hauptschleife relevant sind, eingegangen.

Alle Aktionen, die mit der Wiimote ausgeführt werden, ändern intern eine Menge an Variablen, die für die Berechnung der Positionen der Bausteine genutzt werden. Die Berechnungen werden bei jedem Durchlauf der Schleife neu aufgerufen. Die Berechnungen liefern als Ergebniss verschiedene Werte, z.B. ob das aktuelle Objekt mit bestehenden Objekten kollidiert, oder ob die Grube zu voll ist und somit das Spiel beendet ist. Ob das Spiel beendet ist, wird mit der Variablen *gameover* überprüft und bei Erfolg wird das Programm pausiert und auf Reaktivierung durch den Benutzer gewartet. Wenn das Spiel noch nicht beendet ist, wird geprüft ob der gesetzte Stein eine Linie füllt und diese dann aus dem Spiel entfernt. Alle Steine die oberhalb dieser Linie sind, werden dann die Anzahl an Linien die entfernt wurden, nach unten geschoben. Danach wird die komplette Szene neu gezeichnet.

[6] http://abstrakraft.org/cwiid, CWiid ist eine Sammlung von Linux Werkzeugen die in C geschrieben sind, um eine Verbindung zur Wiimote zu verwalten. Sie enthält eine Event basierende API.

Programmablaufplan von Tetris GL

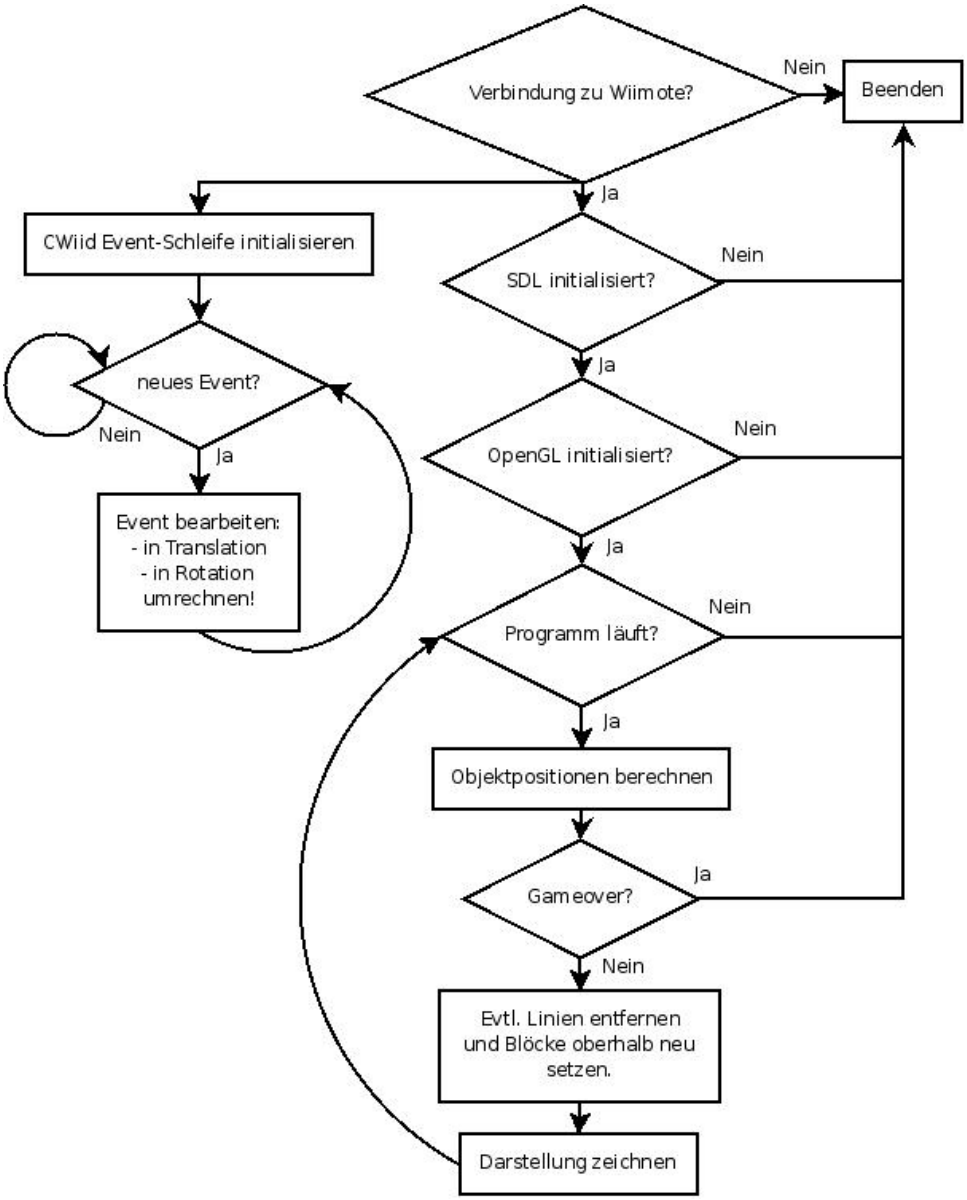

Abbildung 10: Vereinfachter Programmablaufplan

6 Konzept

Im folgenden wird das Umsetzungskonzept von **Tetris GL** erläutert.

In Abbildung 11 wird die Abhängigkeit von den Klassen die für die Bausteine verwendet werden, verdeutlicht. Die Basis aller Bausteine bildet die Klasse **glObject**, die zahlreiche Standardinformationen enthält. Dies sind u.a. die Position der Bausteine im 3D Raum,

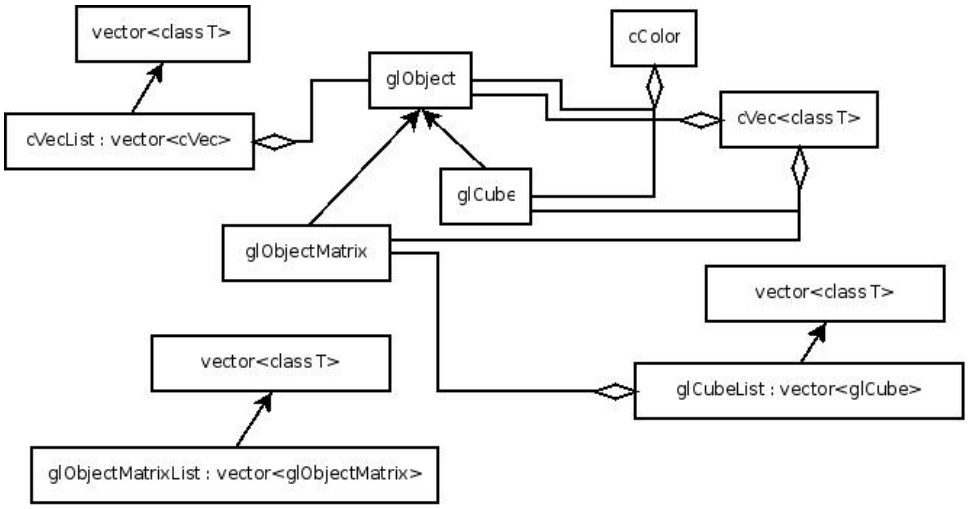

Abbildung 11: Klassenabhängigkeit von Tetris GL. (UML Konvention)

die Werte der BoundingBox[7], die Bewegungsrichtung, die aktuelle Rotation, die Farbe, die Punktkoordinaten der Ecken eines Objektes und die eindeutige Identifikationsnummer einer Textur im Grafikspeicher, die auf die Bausteine gemapped wird.

Die Klasse **glCube** ist für die ordentliche Darstellung und Berechnung der Koordinaten im 3D Raum zuständig. Alle Koordinaten, die an diese Klasse übergeben werden, müssen als relative Werte zum Nullpunkt im Raum bei der Koordinate 0, 0, 0 für x, y und z angegeben sein. Bei jedem Zeichnen der einzelnen Cubes werden die Koordinaten neu berechnet, egal ob es eine Änderung gab. Im Kapitel 8 wird darauf eingegangen. Diese Klasse beinhaltet alle OpenGL-Aufrufe.

glCubeList: Diese Klasse kann eine beliebige Anzahl an glCube-Objekten aufnehmen und erbt alle Methoden der Template Container Klasse vector aus der libstdc++ Bibliothek, die bei jedem Linux System automatisch mitgeliefert wird. Der Zugriff benötigt eine lineare Zeit, in Abhängigkeit mit der Anzahl der Elemente, die sich in dem Container befinden.

Die Klasse **glObjectMatrix** beinhaltet ein raffiniertes Konzept, das es dem Entwickler ermöglicht, auf dem schnellsten Wege neue Bausteine in das Spiel zu integrieren. Listing 2 liefert ein Beispiel, wie neue Bausteine hinzugefügt werden.

```
1  #include "forms.h"
2  #include "cColor.h"
3  #include "glCube.h"
4  #include "glObjectMatrix.h"
5  ...
6
7  /** Die Einsen geben an, an welcher Position, sich ein Kubus (Cube)
       befindet.
8   */
9  static int __rechteck[] = {
10         0, 0, 1,
11         1, 1, 1
12 };
13
14 enum FORMSID { FRM_RECT };
15
16 /** Struktur, die alle Bausteine beinhaltet.
```

[7]Eine/Ein Bounding Box/Bounding Volume ist in der algorithmischen Geometrie ein einfacher geometrischer Körper, der ein komplexes dreidimensionales Objekt oder einen komplexen Körper umschließt.

```
17    */
18   static struct forms {
19           int * frm;
20           int cols;
21           int rows;
22           int depth;
23           cColor color;
24           FORMSID id;
25           int points;
26   } forms[1] = {
27    { __rechteck, 3, 2, 1, cColor(COLOR_RED), FRM_RECT, 50 }
28   };
29
30   ...
31   /** Hinzufuegen des oben definierten Bausteines.
32    */
33   Tetris::glObjectMatrix baustein;
34
35   struct forms * f = &forms[0];
36
37   baustein.position() = cVec<float>(0.0f, 20.0f, 0.0f);
38
39   baustein.setMatrix( f->frm, f->cols, f->rows,
40                       f->depth, f->color, f->id, 0 );
41
42   baustein.rotateX(90.0f);
43   baustein.rotateY(90.0f);
44   baustein.rotateZ(90.0f);
45
46   baustein,setCubeSectorArea(
47    cVec<float>( PLAYGROUND_STARTX, PLAYGROUND_STARTY, PLAYGROUND_STARTZ ),
48    cVec<float>( PLAYGROUND_STARTX, PLAYGROUND_STARTY, PLAYGROUND_STARTZ ),
49    1.0f );
50
51   ...
```

Listing 2: Beispiel für das Hinzufügen von Bausteinen in das Spiel

Zeilen 1-4 inkludieren alle für dieses Beispiel, wichtigen Headerdateien. Diese enthalten die Definitionen der Klassen und Bausteinelemente.

In Zeile 9 wird ein einfaches Bausteinelement definiert, das dem Rechtwinkel Objekt, Abbildung 5 in Kapitel 3, entspricht. Zeile 14 definiert eine eindeutige Identifikations-nummer für das Element. Zwischen den Zeilen 18 und 28 wird die *forms* Struktur definiert, die alle Bausteinelemente erhält.

Zeile 33 erstellt ein Objekt, welches das Rechteckelement aufnimmt, was die Zeilen 39 und 40 beschreiben. Die Methode *setMatrix* des *baustein* Objektes erwartet als Parameter die beschreibenden Daten der *forms* Struktur. Der letzte Parameter ist die Identifikationsnummer einer Textur, die schon zuvor in den Grafikspeicher geladen wurde. Sollte eine Null übergeben werden, wird keine Struktur auf die Bausteine gemapped, sondern nur eine Farbe die zufällig erstellt wird.

In Zeile 37 wird die Startposition des Bausteins im 3D Raum angegeben. Die Koordinaten werden für die X-, Y- und Z-Achse angegeben.

Die Methoden zum rotieren sind in diesem Listing nur als exemplarisches Beispiel eingefügt und müssen an dieser Stelle nicht verwendet werden.

In Zeile 46 bis 49 wird der Kollisionsbereich angegeben. Dies sind die minimalen und maximalen Koordinaten des Bereiches in dem sich die Benutzerinteraktion ereignet. Alle Bewegungen und Änderungen der Bausteinpositionen befinden sich in diesem Bereich.

Der erste Parameter ist der minimale Wert, der zweite Parameter ist der maximale Wert, der letzte Parameter ist die Schrittweite, diese teilt den Raum in eine m mal n Matrix.

glObjectMatrixList: Diese Klasse kann eine beliebige Anzahl an glObjectMatrix Objekten aufnehmen und erbt alle Methoden der Template Container Klasse vector aus der libstdc++ Bibliothek, die bei jedem Linux System automatisch mitgeliefert wird.

cVec<class T>: Eine Klasse zum Berechnen von mathamtischen Vektoren, die mehrere Operatoren überlädt um auf einfachste Weise mit Berechnungen umgehen zu können.

cVecList<class T>: Siehe dazu *glCubeList* und *glObjectMatrixList*.

cColor: Diese Klasse repräsentiert eine Farbe in RGB Darstellung.

7 Implementierungen

In diesem Kapitel wird auf verschiedene Funktionen eingegangen, die Einblick in die Funktionalität des Spiels bringen.

Grundidee: Alle Bausteine bestehen aus einer Anzahl von glCube Objekten, wie in Abbildung 11 zu sehen. Der gesamte Spielraum ist in Sektoren unterteilt, siehe Listing 2 in Kapitel 6 und Abbildung 12.

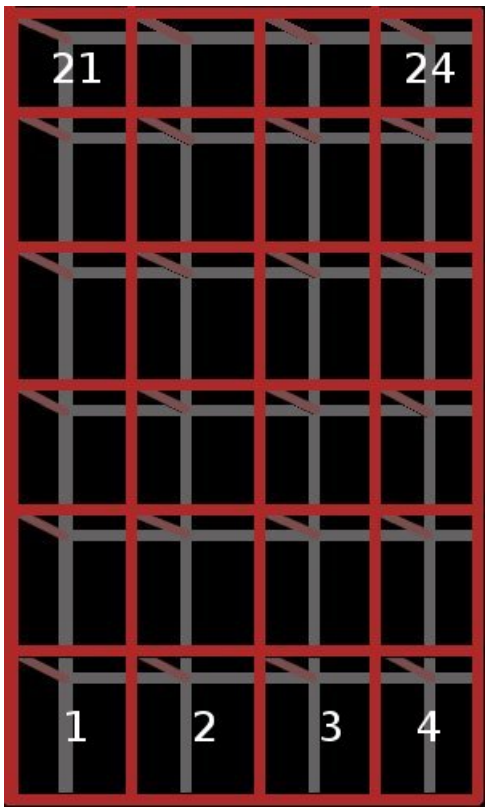

Abbildung 12: Aufteilung des Spielraumes in eine Gitterbox, examplarische Darstellung

Beim Setzen eines Bausteins werden alle glCube Objekte dieses Bausteins auf die aktuellen Sektoren aktualisiert, in denen sich der Baustein befindet. Wenn das Spielfeld eine

Größe von 10x10 (X- mal Z-Achse) hat, dann sind die Sektorenwerte von eins bis zehn. Der erste Sektor ist dann X=1 und Z=1 und der letzte Sektor ist X=10 und Z=10. Bei der Kollisionserkennung werden die Werte der BoundingBox jedes einzelnen glCube Objektes mit den glCube Objekten des aktuellen Bausteins verglichen. Sollte keine Kollision vorhanden sein, wird der Stein in die Richtung, in die er sich bewegen soll, weiter berechnet. Siehe dazu Kapitel 7.1.

Beim Berechnen von komplett gesetzten Linien und der Berechnung von den neuen Positionen von evtl. oberhalb befindlichen glCube Objekten der Bausteien werden die Sektoren verwendet. Alle Steine oberhalb eines entfernten glCube Objektes werden um einen Sektor nach unten verschoben. Siehe dazu Kapitel 7.2.

7.1 Bewegungsfunktionen / Kollisionserkennung

In den Dateien movement.h und movement.cpp sind fünf Funktionen definiert und implementiert, die eine sehr wichtige Funktion erfüllen. Diese Funktionen dienen der Bewegungsberechnung der Bausteine und überprüfen ob Kollisionen im 3D Raum stattfinden. Es wird im folgenden auf vier von diesen Funktionen eingegangen, es werden nur vier beschrieben, da sich die fünfte sehr mit einer vorhandenen ähnelt.

- **void move_x(**
  float step,
  Tetris::glObjectMatrix & current_object,
  Tetris::glObjectMatrixList & cube_list
  );

  Siehe dazu Listing 6. Bewegt das Objekt in negative/positive Richtung auf der X-Achse.

  step: Dieser Parameter ist die Schrittweite des Objektes, um die es sich bewegen soll.

  current_object: Ist das aktuelle Objekt, welches sich im Interaktionsbereich des Benutzers befindet. Alle Aktionen werden auf dieses Objekt angewendet.

  cube_list: Beinhaltet alle Bausteine, die bisher im Spiel gespielt wurden bzw. die schon gesetzt wurden.

- **void move_z(**
  float step,
  Tetris::glObjectMatrix & current_object,
  Tetris::glObjectMatrixList & cube_list
  );

  Funktion: Diese Funktion ähnelt der move_x Funktion.

- **int check_xyz(**
  CHKMODE m,
  Tetris::glObjectMatrix & current,
  Tetris::glObjectMatrixList & cube_list,
  float ystep = 0.0f
  );

  check_xyz(...) (siehe dazu Kapitel 7.1.1) übernimmt die Prüfung für die einzelnen Achsen im 3D Raum.

m Dies ist ein enum Wert, dieser gibt an auf welcher Achse die Kollision überprüft werden soll. Die möglichen Werte stehen in der Datei „movement.h".

current: Aktuelles Objekt, das überprüft werden soll.

cube_list: Die Liste alle Objekte, die mit current verglichen werden.

ystep: Dieser Wert bestimmt, wie weit das Objekt auf der Y-Achse nach unten gesetzt wird, wenn keine Kollision aufgetreten ist.

- **bool check_gameover(**
  Tetris::glObjectMatrix & current,
  float maxy
  );

  Diese Funktion überprüft, ob ein Stein nicht weiter nach unten auf der Y-Achse gesetzt werden kann. Wenn dies der Fall ist, wird TRUE zurück geliefert und das Spiel ist beendet.

  current_object: Ist das aktuelle Objekt, welches sich im Interaktionsbereich des Benutzers befindet. Die Überprüfung wird auf dieses Objekt angewendet.

  maxy: Die maximale Koordinate auf der Y-Achse. Wenn ein Stein diese Grenze überschreitet, dann ist das Spiel beendet, da dieser nicht weiter nach unten gesetzt werden kann.

- **MUSICTRACK calculate_objects(**
  float ss,
  Tetris::glObjectMatrix & current_object,
  Tetris::glObjectMatrix & next_object,
  Tetris::glObjectMatrixList & cube_list
  );

  calculate_objects (siehe Listing 7) enthält die Logik, die mit den oben genannten Funktionen die neuen Positionen der Bausteinobjekten berechnet und für die Objekte setzt.

  ss Die Schrittweite, die im passenden Verhältnis zur Framerate für das aktuelle Bausteinobjekt verwendet werden soll.

  current_object: Ist das aktuelle Objekt, welches sich im Interaktionsbereich des Benutzers befindet. Die Berechnung wird auf dieses Objekt angewendet.

  next_object: Das Objekt, welches beim Setzen des aktuellen Objektes („current_object") zum neuen aktuellen wird.

  cube_list: Die Liste aller Objekte, die mit current_object verglichen werden.

7.1.1 check_xyz(...)

Im folgenden wird auf diese Funktion näher eingegangen. Die Beschreibungen stehen als Kommentare im Quelltext.

enum-Werte von CHKMODE

```
1 enum CHKMODE {   CHK_NONE=0x00,
2                  CHK_X=0x01,
3                  CHK_Y=0x02,
4                  CHK_XY=CHK_X|CHK_Y,
5                  CHK_Z=0x04,
```

```
6                   CHK_XZ=CHK_X|CHK_Z,
7                   CHK_YZ=CHK_Y|CHK_Z,
8                   CHK_ALL = CHK_X | CHK_Y | CHK_Z
9 };
```

Listing 3: C++ enumerate Werte von CHKMODE.

Die Funktion check_xyz(...) im Detail

```
1  int check_xyz( CHKMODE m,
2          Tetris::glObjectMatrix & current,
3          Tetris::glObjectMatrixList & cube_list,
4          float ystep )
5  {
6    // Der Rueckgabewert der Funktion.
7    int ret = CHK_NONE;
8
9  /* Dieser Wert ist ein Abstandsmass fuer die Ueberpruefung der Kollision
10  * zwischen dem aktuellen Baustein und den schon im Spiel vorhandenen.
11  * Die untere Skizze soll dies verdeutlichen.
12  */
13 #define V 0.02f
14
15   /* Alle glCube Objekte eines Bausteins durchgehen und
16    * auf Kollision mit dem aktuellen Bausteinobjekt ueberpruefen.
17    */
18   for( int i = 0; i < current.cubeCount(); i++ )
19   {
20 #define CURRENT current.cubes()->at(i)
21
22     float x_c_low = CURRENT.lowPosition().x();
23     float y_c_low = CURRENT.lowPosition().y();
24     float z_c_low = CURRENT.lowPosition().z();
25
26     float x_c_high = CURRENT.highPosition().x();
27     float y_c_high = CURRENT.highPosition().y();
28     float z_c_high = CURRENT.highPosition().z();
29
30     // Sollte der Stein tiefer sein, als die Spielflaeche,
31     // dann muss keine weitere Berechnung durchgefuehrt werden.
32     if( y_c_low <= 0.0f ) {
33       return CHK_ALL;
34     }
35
36     x_c_low += V; x_c_high -= V;
37     y_c_low += V; y_c_high -= V;
38     z_c_low += V;   z_c_high -= V;
39
40     vector<Tetris::glObjectMatrix>::iterator it = cube_list.begin();
41     for( ; it != cube_list.end(); it++ )
42     {
43       for( int j = 0; j < it->cubeCount(); j++ ) {
44 #define CHECK it->cubes()->at(j)
45
46         if( m & CHK_X ) {
47           float x_low = CHECK.lowPosition().x();
48           float x_high = CHECK.highPosition().x();
49
50           // Ueberprueft ob die BoundingBox-Werte der X-Achse
51           // im Bereich einer anderen BoundingBox kollidieren.
52           if( x_c_low >= x_low && x_c_low <= x_high )
53             ret |= CHK_X;
54         }
55
56         if( m & CHK_Z ) {
57           float z_low = CHECK.lowPosition().z();
58           float z_high = CHECK.highPosition().z();
```

```
59
60            // Ueberprueft ob die BoundingBox-Werte der Z-Achse
61            // im Bereich einer anderen BoundingBox kollidieren.
62            if( z_c_low >= z_low && z_c_low <= z_high )
63              ret |= CHK_Z;
64          }
65
66          // Nach einer Kollision ist das aktuelle Objekte tiefer als das
67          // Objekt, mit dem es kollidiert ist, drum muss es verschoben
68          // werden. Dieses Delta ist der Wert, um das es verschoben wird.
69          float dy = 0.0f;
70
71          if( m & CHK_Y ) {
72            float y_low = CHECK.lowPosition().y();
73            float y_high = CHECK.highPosition().y();
74
75            dy = y_c_low - y_high;
76
77            float dd = y_low - y_c_high;
78
79            if( dy < fabs(ystep) && dd < 0.0f )
80              ret |= CHK_Y;
81          }
82  #undef CHECK
83          if( ret == m )
84          {
85            for( int index = 0; index < current.cubeCount(); index++ )
86              {
87  #define L current.cubes()->at(index).lowPosition()
88  #define H current.cubes()->at(index).highPosition()
89  #define P current.cubes()->at(index).position()
90                  L.y() += dy - 1.0f;
91                  H.y() += dy - 1.0f;
92                  P.y() += dy;
93  #undef P
94  #undef H
95  #undef L
96              }
97            return ret;
98          } else
99            ret = CHK_NONE;
100       }
101     }
102 #undef CURRENT
103   }
104 #undef V
105   return ret;
106 }
```

Listing 4: Detailbeschreibung von der Funktion check_xyz(...).

Kollisionerkennung des aktuellen Bausteins

In Abbildung 13 ist die Kollisionerkennung des aktuellen Bausteins mit anderen schon gesetzten Bausteinen verdeutlicht. Der BoundingBox Werte des aktuellen Bausteins werden manuell um einen Wert (siehe Listing 4) geschrumpft. Nun wird überprüft, ob sich diese Werte im Bereich eines anderen Bausteines befinden. Diese Überprüfung ist ohne Probleme im 3D Raum, für drei Achsen möglich und verhindert auch, dass es bei dieser Überprüfung zu Rundungsfehlern kommt.

Es hat sich im Verlauf der Entwicklung dieses Spiels gezeigt, dass es bei der Subtraktion eines Wertes auf der Y-Achse, um den Stein nach unten zu verschieben, zu Rundungsfehlern kommt. Dies hat mit der Darstellung von Fließkommazahlen im Hauptspeicher zu tun. Da die Genauigkeit in der Abspeicherung auf eine bestimmte Speichergröße beschränkt ist.

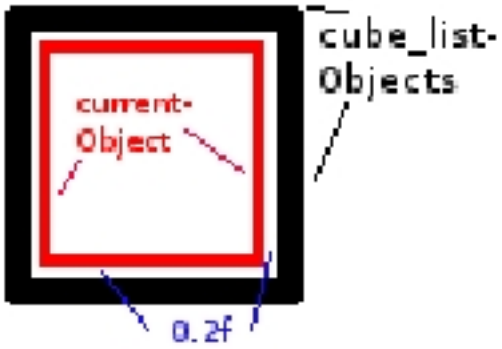

Abbildung 13: Kollisionsgrafik für den aktuellen Baustein

Siehe dazu Listing 5. In der Ausgabe ist unschwer zu erkennen, dass nicht der erwartete Wert aus-gegeben wird. Um diesen Fehler zu entgehen, bietet es sich an, die oben genannte Technik zu verwenden.

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main( int argc, char **argv ) {
6          float fz = 0.5f - 0.1f; // Sollte 0.4f als Ergebniss liefern!
7          cout << "fz: " << setprecision(32) << fz << endl;
8  }
9
10 // Ausgabe:
11 fz: 0.40000000596046447753906250
```

Listing 5: Rundungsfehler bei Fließkommazahlen

7.1.2 move_x(...)

Im folgenden wird auf diese Funktion näher eingegangen. Die Beschreibungen stehen als Kommentare im Quelltext.

```
1  void move_x( float step,
2            Tetris::glObjectMatrix & current_object,
3            Tetris::glObjectMatrixList & cube_list )
4  {
5    // Das aktuelle Objekt wird i ein temporaeres Objekt kopiert, dass
6    // die Koordinaten der neuen Position enthaelt. Auf dieses Objekt
7    // wird die Kollisionserkennung angewandt. Sollte keine Kollision
8    // vorhanden sein, wird die neue Position uebernommen.
9    Tetris::glObjectMatrix objTmp(current_object);
10
11   Tetris::glCubeList * c = objTmp.cubes();
12
13   for( int i=0; i<objTmp.cubeCount(); i++ )
14   {
15 #define C current_object.cubes()->at(i)
16     c->at(i).lowPosition().x() = C.lowPosition().x() + step;
17     c->at(i).highPosition().x() = C.highPosition().x() + step;
18
19     c->at(i).lowPosition().y() = C.lowPosition().y();
20     c->at(i).highPosition().y() = C.highPosition().y();
21
```

```
22      c->at(i).lowPosition().z() = C.lowPosition().z();
23      c->at(i).highPosition().z() = C.highPosition().z();
24  #undef C
25    }
26
27    // Ueberprueft auf allen drei Achsen ob eine Kollision vorhanden ist.
28    int ret = check_xyz( CHK_ALL, objTmp, cube_list );
29
30    // Sollte step ein negativer Wert sein, dann soll das Objekt in negative
31    // Richtung auf der X-Achse verschoben werden. Sollte das Spielfeldende
32    // erreicht sein, wird eine Kollision vorgetaeuscht.
33    // Selbes gilt fuer positive Richtung.
34    if( step < 0 )
35    {
36      if( current_object.lowPosition().x() + step < PLAYGROUND_STARTX ||
37          ret == CHK_ALL )
38        return;
39    } else
40    {
41      if( current_object.highPosition().x() + step > PLAYGROUND_ENDX ||
42          ret == CHK_ALL )
43        return;
44    }
45
46    // Die neue Position des Bausteinobjektes wird berechnet.
47    current_object.lowPosition().x() += step;
48    current_object.highPosition().x() += step;
49    current_object.position().x() += step;
50  }
```

Listing 6: Detailbeschreibung von der Funktion move_x(...)

7.1.3 calculate_objects(...)

Im folgenden wird auf diese Funktion näher eingegangen. Die Beschreibungen stehen als Kommentare im Quelltext.

```
1  MUSICTRACK calculate_objects( float ss,
2                                Tetris::glObjectMatrix & current_object,
3                                Tetris::glObjectMatrix & next_object,
4                                Tetris::glObjectMatrixList & cube_list)
5  {
6    // Auf allen drei Achsen, auf Kollisionen, ueberpruefen.
7    int ret = check_xyz( CHK_ALL, current_object, cube_list, ss );
8
9    if( ret == CHK_ALL ) {
10     float dy = current_object.lowPosition().y();
11     if( dy < 0.0f )
12     {
13       Tetris::glCubeList::iterator sit = current_object.cubes()->begin();
14       Tetris::glCubeList::iterator eit = current_object.cubes()->end();
15
16       for( ; sit != eit; sit++ ) {
17         sit->lowPosition().y() += dy;
18         sit->highPosition().y() += dy;
19         sit->position().y() += dy;
20       }
21     }
22     // Folgende Anweisung sind fuer die Musikwiedergabe zustaendig.
23     // Beim Setzen eines Steines wird ein Wert zurueckgeliefert, der
         aussagt,
24     // das eine zu dieser Aktion passende Musikdatei abgespielt werden soll
         .
25     if(play_music_set == false )
```

```
26        return SET;
27      else
28        return NONE;
29    } else {
30      // Position des Objektes auf der Y-Achse neu setzen.
31      current_object.position().y() += ss;
32    }
33
34    return NONE;
35  }
```

Listing 7: Dateilbeschreibung von der Funktion calculate_objects(...)

7.2 Logische Funktionen

In den Dateien logic.h und logic.cpp sind zwei Funktionen definiert und implementiert, die die logische Verarbeitung der gesetzten Bausteine übernehmen. Die Aufgaben sind unter anderem das Erkennen und Entfernen von kompletten Zeilen und Entfernen von glObject-Matrix Objekten aus der Liste aller gesetzten Bausteine, im Grunde eine Speicher-freigabe von nicht genutzen Speicher.

- **void remove_empty_objects(**
  Tetris::glObjectMatrixList & cube_list
  ); Diese Funktion entfernt nicht genutzte glObjectMatrix Objekte aus der Liste aller bisher gesetzten glObjectMatrix Objekte. Dies sind alle, die keine glCube Objekte mehr besitzen.

  cube_list: Liste aller gesetzten Bausteinobjekte.

- **void remove_line(**
  Tetris::glObjectMatrix & current_object,
  Tetris::glObjectMatrixList & cube_list,
  cVecList<int> * x_list, cVecList<int> * z_list
  ); Diese Funktion entfernt alle vervollständigten Linien auf der X- und Z-Achse. Die Sektoren der glCube Objekte, die entfernt werden, werden in den Listen x_list und z_list abgespeichert.

  current_object: Aktuelles Bausteinobjekt.

  cube_list: Liste aller bisher gesetzten Bausteinobjekte.

  x_list: Die Sektorenwerte aller glCube Objekte, die entfernt wurden.

  z_list: Siehe x_list!

7.2.1 remove_line(...)

Im folgenden wird auf diese Funktion näher eingegangen. Die Beschreibungen stehen als Kommentare im Quelltext.

```
1  void remove_line( Tetris::glObjectMatrix & current_object,
2                    Tetris::glObjectMatrixList & cube_list,
3                      cVecList<int> * x_ret_list, cVecList<int> * z_ret_list
4              )
5  {
6    /// Letzten gesetzten Cube durchgehen.
7    Tetris::glCubeList::iterator cc_it = current_object.cubes()->begin();
```

```
8     for( ; cc_it != current_object.cubes()->end(); cc_it++ )
9     {
10 #define CCSEC cc_it->sector()
11
12      // Liste der Sektoren von glCube Objekten, die entfernt werden.
13      cVecList<int> x_list;
14      cVecList<int> z_list;
15
16      // Alle bisher gesetzten Cubes durchgehen und deren ...
17      Tetris::glObjectMatrixList::iterator cl_it = cube_list.begin();
18      for( ; cl_it != cube_list.end(); cl_it++ )
19      {
20        // ... einzelene Cubes auf Achsengleichheit ueberpruefen.
21        for( int i = 0; i < cl_it->cubeCount(); i++ )
22        {
23 #define CLSEC cl_it->cubes()->at(i).sector()
24          // Auf der X-Achse ueberpruefen.
25          if(   CLSEC.y() == CCSEC.y() &&
26              CLSEC.z() == CCSEC.z()
27              )
28            x_list.push_back(CLSEC);
29
30          // Auf der Z-Achse ueberpruefen.
31          if(   CLSEC.y() == CCSEC.y() &&
32              CLSEC.x() == CCSEC.x()
33              )
34            z_list.push_back(CLSEC);
35 #undef CLSEC
36        }
37      }
38
39      // Ueberpruefen ob eine Zeile auf der X-Achse vollendet wurde.
40      // Das Spielfeld hat eine Breite von 10 Sektoren.
41      if( x_list.size() == 10 )
42      {
43        x_ret_list->assign( x_list.begin(), x_list.end() );
44
45        cl_it = cube_list.begin();
46        Tetris::glObjectMatrixList::iterator cl_it_end = cube_list.end();
47        for( ; cl_it != cl_it_end; cl_it++ )
48        {
49            for( int ii = 0; ii < x_list.size(); ii++ )
50            {
51              cl_it->removeCubeX(x_list.at(ii));
52            }
53        }
54
55        if( current_object.cubeCount() <= 0 )
56          return;
57      }
58
59      // Ueberpruefen ob eine Zeile auf der Z-Achse vollendet wurde.
60      // Das Spielfeld hat eine Breite von 10 Sektoren.
61      if( z_list.size() == 10 )
62      {
63        z_ret_list->assign( z_list.begin(), z_list.end() );
64
65        cl_it = cube_list.begin();
66        Tetris::glObjectMatrixList::iterator cl_it_end = cube_list.end();
67        for( ; cl_it != cl_it_end; cl_it++ )
68        {
69          for( int ii = 0; ii < z_list.size(); ii++ )
70          {
71            cl_it->removeCubeZ(z_list.at(ii));
72          }
73        }
74      }
```

```
75 #undef CCSEC
76   }
77 }
```

Listing 8: Detailbeschreibung von der Funktion remove_line(...)

7.3 Hauptschleife

Dieses Kapitel beschreibt die Hauptschleife des Spiels, in der alle oben genannten Funktionen aufgerufen werden und in der bestimmt wird, ob u.a. das Spiel pausiert ist oder beendet und wie die Musik abgespielt wird.

7.3.1 main -> Loop-Game

```
1  while( run ) {
2    // Ein boolean-Wert, der angibt ob Hintegrundmusik gespielt werden soll.
3    if(play_music)
4      Mix_Playing(channel_theme);
5
6    // Frames per Second (FPS) Berechnung, wichtig um auf verschiedenen
7    // Rechnern, die unterschiedliche Ressourcen haben, einen zeitlich
8    // identischen Spielverlauf zu haben.
9    idle_func();
10
11   // Bearbeitet die Eingaben des Benutzers (Tastatureingaben).
12   process_events();
13
14   // Das Spiel weiterlaufen lassen, wenn es nicht zu ende oder
15   // pausiert ist.
16   if(!gameover || !paused)
17   {
18     // Positionen der Bausteinobjekte berechnen.
19     int r = calculate_objects(    step_fps(),
20                           current_object,
21                           next_object,
22                           cube_list );
23
24     // Der Rueckgabewert r gibt an was zu tun ist, ob ein Stein
25     // gesetzt wurde oder ob dieser nur bewegt wurde.
26     switch(r)
27     {
28       case SET: {
29         // Ist das Spiel beendet, nach dem naechsten Zug?
30         gameover = check_gameover( current_object, wuerfel_start.y() );
31         // Sektoren neu setzen.
32         current_object.updateSectors();
33
34         Tetris::glObjectMatrix co = current_object;
35
36         // Neues Bausteinobjekt erzeugen.
37         current_next_cube( current_object, next_object, cube_list );
38
39         cVecList<int> x_list, z_list;
40
41         // Die Liste der Sektoren ermitteln die entfernt wurde.
42         remove_line( co, cube_list, &x_list, &z_list );
43
44         // Nicht mehr benoetigte Bausteinobjekte aus der Bausteinliste
               entfernen.
45         remove_empty_objects( cube_list );
46
```

```
47          // Die Steine oberhalb der entfernten Sektoren um einen Sektor nach
                unten verschieben.
48          Tetris::glObjectMatrixList::iterator sit = cube_list.begin();
49          Tetris::glObjectMatrixList::iterator eit = cube_list.end();
50          for( ; sit != eit; sit++ )
51          {
52            for( int i=0; i < x_list.size(); i++ )
53            {
54              sit->updateCubeY( x_list.at(i), 0 );
55            }
56
57            for( int i=0; i < z_list.size(); i++ )
58            {
59              sit->updateCubeY( z_list.at(i), 1 );
60            }
61          }
62
63          // Je nachdem ob ein Stein gesetzt wurde, eine bestimmte Sounddatei
                abspielen.
64          if(play_music)
65          {
66              play_music_set = true;
67              Mix_ChannelFinished(my_audio_finished);
68              Mix_Chunk *sound =
69                  Mix_LoadWAV("data/music/block-set1.wav");
70              if(sound == NULL) {
71                fprintf(stderr, "Unable to load WAV file: %s\n",
72                    Mix_GetError());
73              }
74              channel_set = Mix_PlayChannel(-1, sound, 0);
75              if(channel_set == -1) {
76                fprintf(stderr, "Unable to play WAV file: %s\n",
77                    Mix_GetError());
78              }
79          }
80
81        } break;
82        case NONE:
83        default:
84          ;
85      }
86    } // gameover
87
88    // Die komplette Szenerie neu zeichnen.
89    draw_screen();
90  } // while(run)
```

8 Verbesserungswürdig

- Die Steuerung der Nintendo Wii Fernbedienung könnte durch eine Skalierung, die vom Spieler durchgeführt wird, an seine Bedürfnisse angepasst werden.
- Die Performance kann stark verbessert werden, u.a. besitzt die Kollisionserkennung einen erheblichen Aufwand an Rechenzeit. Je mehr Steine in der Liste der gesetzten Bausteine vorhanden sind, je höher ist der zeitliche Aufwand um alle Steine auf eine Kollision zu überprüfen.
- Das Entfernen von komplettierten Linien könnte animiert werden.

9 Probleme bei der Entwicklung

- Das größte Problem bei der Entwicklung dieses Spiels, war das nicht Erkennen von den massiven Rundungsfehlern bei Fließkommazahlen. Es wurde davon ausgegangen, dass Subtraktionen und Addition zu keinen Fehlern führen. In der ersten Version wurde die Kollisionerkennung sehr fehleranfällig, es wurde da noch auf einen äquivalenten Wert der BoundingBoxen überprüft. Diesem Thema sollte besondere Aufmerksamkeit zugesprochen werden.

- Ein weiteres Problem lag an der Programmiersprache C++. Es sollte die **Regel der großen 3** beachtet werden. Diese Regel lautet:

Ist eine der Memberfunktionen - Kopierkonstruktor - Destruktor - Überladener Zuweisungsoperator notwendig, so sind in der Regel auch die beiden anderen Memberfunktionen erforderlich.

  Sollte diese Regel nicht Beachtung finden, kann es sein, dass beim Zuweisen oder Erstellen von Objekten nicht alle Daten des schon erstellten Objektes rüberkopiert werden. Der Programmierer ist dafür verantwortlich, dass der Kopierkonstruktor oder der überladene Zuweisungsoperator definiert wird. Wer sich keine Sorgen machen will, sollte einfach alle oben genannten Methoden implementieren!

Literaturverzeichnis

[1] Wright, Richard S. and Lipchak, Benjamin: OpenGL Superbible, 3. Edition, Sams Publishing, 2005

[2] GPU Gems 2 / Edited by Matt Pharr: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005

[3] Prinz, Peter; Kirch-Prinz, Ulla: C++ Lernen und professionell anwenden, 1. Auflage, Bonn: MITP-Verlag GmbH, 1999

[4] Papula, Lothar; Mathematische Formelsammlung, 8. Auflage, Wiesbaden: Friedr. Vieweg & Sohn Verlag/GWV Fachverlag GmbH

[5] http://nehe.gamedev.net, Neon Helium, Articles and Tutorials about OpenGL