ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption und Realisierung einer auf OpenGL basierenden 3D Welt für eine mobile Multiplayer-Anwendung

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Julian Schweiger
julian.schweiger@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marc Schickler

2014

Fassung 6. November 2014

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Die mobile Appllikation *Track Your Tinnitus* ist ein Forschungsprojekt in dem mittels einer Smartphone App die individuellen Schwankungen der Tinnituswahrnehmung erfasst werden. Diese Erfassung geschieht über die zu beantwortende Fragebögen welche den Zustandes des Tinnitus verdeutlichen. Oftmals klagen die Benutzer nach dieser Befragung, über einen Anstieg des Tinnitus. Um diesen Effekt zu mildern soll der Benutzer im Anschluss des Fragebogens von seinem Tinnitus abgelenkt werden.
Im Rahmen dieser Arbeit wird ein zweidimensionales Labyrinth in einer auf OpenGL basierenden, dreidimensionalen Welt konzipiert und realisiert. Dieses wird, um den Spieler von seinem Tinnitus abzulenken, für die *Track Your Tinnitus* Applikation erweitert und soll diese ergänzen. Darüber hinaus wird das Labyrinth eine Grundlage für ein eigenständiges Spiel schaffen.

Inhaltsverzeichnis

1

Einleitung

Laut eines Bericht des Bundesverbandes für interaktive Unterhaltungssoftware steigt der Umsatz von Unterhaltungssoftware jährlich um 6,5 Prozent [Bun14a]. In Deutschland ist mit mittlerweile 31 Millionen Nutzern von Computer- und Videospiele eine ähnliche Entwicklung erkennbar [Bun14a]. Ein wesentlicher Bestandteil des Computerspielemarktes ist der mobile Sektor geworden. Der Bundesverband für interaktive Unterhaltungssoftware beziffert den Umsatz für mobile Videospiele in Deutschland auf 114 Millionen Euro allein in der ersten Hälfte des Jahres 2014 - Tendenz steigend [Bun14b]. “Spiele-Apps sind der wesentliche Treiber des Smartphone-Booms“ befindet Dr. Maximilian Schenk, Geschäftsführer des BIU [Bun14b]. Mit 48% spielt knapp jeder Zweite zwischen 6 und 19 Jahren mehrmals pro Woche auf dem Smartphone [Bun14b]. Nicht nur zu Hause am PC oder an einer Spiele-Konsole, sondern auch auf mobilen Geräten sind Videospiele allgegenwärtig. Diese Entwicklung hat den Anstoß gegeben die Entstehung eines Videospieles näher zu betrachten. Da mobile Spieleapplikationen weniger komplex

und umfangreich sind als Spiele auf High-End-Geräten, sind sie für den Einstieg in die Thematik der Spieleentwicklung besonders geeignet [GSP+14].
Wir spielen um uns abzulenken und uns zu unterhalten. Videospiele werden als *Serious-Games* zur Verbesserung der kognitiven Eigenschaften eingesetzt und längst haben sie Einzug in das soziale Umfeld genommen. Videospiele lassen uns in eine andere Welt eintauchen und können uns das Umfeld vergessen lassen. Diesen Effekt soll in der App *Track Your Tinnitus* genutzt werden um den Patienten nach der Datenerfassung von seinem Tinnitus abzulenken.

1.1 Ziel der Arbeit

In dieser Arbeit soll sowohl ein Einblick in die Grafikprogrammierung gegeben werden als auch in die Grundlagen zur Entwicklung eines mobilen Videospieles für Android. Zur Veranschaulichung wird eine begehbare dreidimensionale Welt als Basis entwickelt. Diese Welt soll durch begehbare Wege mit einander verbunden und durch nicht durchschreitbare Wände geteilt sein. Dadurch soll ein zweidimensionales Labyrinth, in einer dreidimensionalen Umgebung entstehen. Diese Arbeit wird diese Welt für zwei Anwendungsfälle benutzen: Zum Einen soll eine Erweiterung der *Track Your Tinnitus*-App entstehen, die die maximale Aufmerksamkeit des Benutzers erlangt. Zum Anderen soll ein eigenständiges Spiel entstehen, das eine hohe Einsteigerfreundlickeit und Langzeitmotivation aufweist [SSP+13].

1.2 Aufbau der Arbeit

Die Arbeit ist in sechs Kapitel unterteilt. Zunächst werden in Kapitel 2 einige Grundlagen erklärt, die in dieser Arbeit verwendet werden. Dabei gibt das Kapitel einen Überblick über die Grafikprogrammierung mit *OpenGL*, erörtert Definition und Entstehung von Labyrinthen und erklärt den Begriff Tinnitus und die Applikation *Track Your Tinnitus*. Grundlagen der Informatik und der Javaprogrammierung sowie Grundlagen zur Entwick-

lung einer Android-App werden im Rahmen dieser Arbeit nicht behandelt. In Kapitel 3 werden sowohl die Anforderungen für das zugrunde liegende Spiel, die Anforderungen in Hinblick *Track Your Tinnitus*, als auch die Anforderungen für das eigenständige Spiel definiert. Kapitel 4 befasst sich mit der spezifischen Umsetzung der zuvor definierten Anforderungen. Hier wird die Architektur der Welt, des Spielgeschehens und die Eingliederung in die jeweiligen Kontexte erklärt. Schließlich werden in Kapitel 5 die zu Beginn definierten Anforderungen abgeglichen. Kapitel 6 erhält eine Zusammenfassung der Arbeit und gibt einen Ausblick für Erweiterungen und zukünftigen Nutzen wieder.

2

Grundlagen

In dieser Arbeit wird ein dreidimensionales Labyrinth für das mobile Betriebssystem Android sowohl als eigenständiges Spiel als auch als Erweiterung der mobilen Applikation *Track Your Tinnitus* konzipiert und entwickelt. In dieser Arbeit wird dafür die offene Grafikbibliothek OpenGL verwendet und einige Grundlagen zur Labyrinthentwicklung. In diesem Kapitel soll ein Verständnis für die Architektur und Arbeitsweise von OpenGL vermittelt und dadurch ein zielorientiertes Anwenden der Funktionen veranschaulicht werden. Außerdem sollen theoretische Grundlagen zu Labyrinthen und deren Generierung gezeigt werden. Abschließend wird der Zusammenhang der Arbeit mit der Applikation *Track Your Tinnitus* erläutert.

2.1 OpenGL

Der Grundstein eines Videospieles ist die Spielewelt. Sie beinhaltet Oberflächen und Objekte. Man trennt hier zwischen dem Datengerüst der Welt, die für Interaktionen und Physik benötigt werden und der reinen Darstellung. Für die Darstellung wird in dieser Arbeit OpenGL verwendet. OpenGL ist eine Schnittstellenspezifikation, die eine programmiersprachen- und plattformunabhängige Grafikprogrammierung erlaubt. Dieser Standard wird von der *Khronos - Group* geführt. Diese ist eine Vereinigung von Unternehmen, wie zum Beispiel AMD, NVIDIA und Oracle, die gemeinsam den Standard definieren und erweitern. Die spezifizierten Funktionen der OpenGL-API werden in den Systembibliotheken der jeweiligen Plattformen implementiert.

2.1.1 Architektur

Die Architektur von OpenGL basiert auf verschiedenen Grundprinzipien, die hier erläutert werden sollen. Grundlegend ist, das OpenGL nicht ergebnisorientiert arbeitet, da nicht ein Resultat beschrieben wird das von OpenGL dargestellt werden soll. Vielmehr werden mit mehreren Funktionsaufrufen die nötigen Prozessschritte von OpenGL eingeleitet, sodass durch diese Bearbeitung ein gewünschtes Resultat erscheint. Daher ist ein Verständnis der Arbeitsweise von OpenGl für dessen Anwendung notwendig. Ein weiterer Aspekt von OpenGL ist, dass es nur einfache grafische Primitive zeichnet, aus denen sich erst komplexere Formen erstellen lassen. Die OpenGL Spezifikation unterstützt zwar bis zu 10 verschiedene eckpunktbasierende grafische Primitive, die sich jedoch alle auf Punkte, Linien und Dreiecke zurückführen lassen [Cla97]. Man spricht deshalb auch von OpenGL als Dreieckszeichenmaschine. Ein weiteres Konzept von OpenGL ist die Client-Server Trennung. Dabei sendet der Client, die Anwendung, Befehle an OpenGL, den Server. OpenGL interpretiert diese Befehle und führt sie aus. Dadurch lässt sich OpenGL basierte Anwendungen ebenso in einer Netwerkumgebung implementieren [Cla97]. Zuletzt ist OpenGL ein Zustandsautomat. Das bedeutet, dass OpenGL abhängig vom aktuellen Zustand die übergebenen Primitiven in den Bildspeicher lädt. Deshalb muss vor jedem Bearbeitungsschritt zunächst der Befehl zur Zustandsänderung gegeben

werden. Da der OpenGL Zustand aus 250 Variablen besteht, soll hier die Arbeitsweise von OpenGL an einer vereinfachten *Rendering-Pipeline* erläutert werden.

2.1.2 Rendering-Pipeline

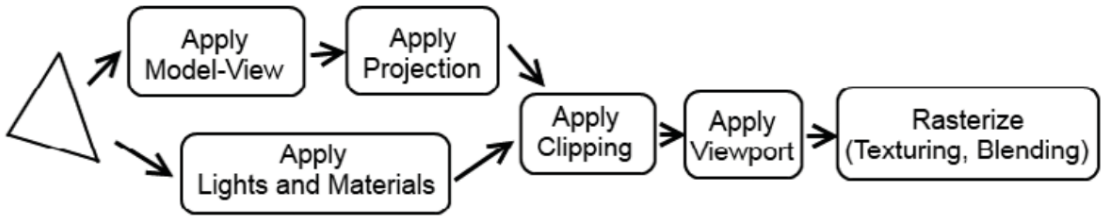

Abbildung 2.1: Vereinfachte Rendering Pipeline [Zec11]

In Abbildung 2.1 ist die sogenannte *Rendering-Pipeline* in stark vereinfachter Form visualisiert. Man erkennt, nach einem zu zeichnenden Dreieck, i Allgemeinen ein *Mesh* (Vgl. Abschnitt 2.1.3, 2.1.3.1), die Zustände, die in OpenGL durchlaufen werden, bevor das Dreieck auf dem Bildschirm ausgegeben werden kann [Zec11].

1. Das Dreieck wird in der Model-View Matrix transformiert(Vgl. Abschnitt 2.1.4.1.
2. In der Projektionsmatrix wird das Ergebnis auf eine zweidimensionale Scheibe projiziert (Vgl. Abschnitt 2.1.4.2).
3. Parallel zu Schritt 1 und 2 wird die Farbe der Pixel berechnet, abhängig von der Grundfarbe, von der Lichtart und dem Lichteinfall, sowie der Materialeigenschaften des Meshes (Vgl. Abschnitt 2.1.5).
4. Es wird die View Frustum berechnet (Vgl. Abschnitt 2.1.4.2).
5. Danach werden die Koordinaten des projizierten Dreiecks in Koordinaten für den FrameBuffer umgerechnet.

2.1.3 Das Mesh

Die zu zeichnenden Objekte der Welt nennt man Meshes. Das Mesh bezeichnet im Kontext der Grafikprogrammierung ein Polygonnetz. Dieses Polygonnetz bildet sich über Ecken und Kanten sowie den Oberflächen des Polygons. Indem man OpenGL Informationen zu drei Punkten im Raum übergibt, kann man OpenGL veranlassen ein Dreieck zu zeichnen. OpenGL errechnet dabei selbst die dazugehörige Oberfläche sowie die Kanten. Durch Verbindung mehrerer Dreiecke kann man so komplexere Formen darstellen.

2.1.3.1 Vertex und Vertices

Ein Vertex (plural: Vertices) bezeichnet in der Computergrafik einen Eckpunkt eines Meshes. Mit Vertices werden in OpenGL die darzustellenden Polygonnetze definiert. Ein Vertex beinhaltet Informationen über die Position im Raum. Die Position der Vertices ist allerdings nur relativ zum Ursprung, da die absolute Position im Raum durch Matrix Projektionen geändert werden kann (Vgl. Abschnitt 2.1.4.1). Ebenfalls kann ein Vertex Informationen über seinen Farbraum und seine Transparenz besitzen.
Da in dieser Arbeit allerdings ausschließlich mit Texturen gearbeitet wird, wird dies vernachlässigt.
Positionsinformationen eines Vertex werden als dreidimensionaler Vektor angegeben. Für ein Mesh werden mindestens drei Vertices benötigt. Damit OpenGL die Vertices interpretieren kann, werden die Vertices zur Übergabe in einen *direct FloatBuffer* [Hit02] geschrieben. Für die Übergabe des *Buffers* an OpenGL, wird OpenGL zunächst in den entsprechenden Zustand gesetzt. Anschließend kann OpenGL mitgeteilt werden, wo der Buffer zu finden ist.

Listing 2.1: Setzen des OpenGL-Zusatndes und des *VertexPointers*

```
1 glEnableClientState(GL_VERTEX_ARRAY);
2 glVertexPointer(3, GL_FLOAT, 0, vertices);
```

2.1.3.2 Texture Mapping

Mit Dreiecken alleine wird man schwer eine lebhafte Welt erstellen können. Um Objekte in der Welt darzustellen, müssen die Oberflächen der Grundgerüste, die Meshes, mit Bildern überzogen werden. Diesen Vorgang nennt man Mapping. Die Bilder mit der die Objekte überzogen werden, nennt man Texturen. Das Texture Mapping ist also die Abbildung von Texturen auf die Meshes.
Eine Textur hat im Kontext OpenGL nun ein eigenes zweidimensionales Koordinatensystem (s,t) mit der Ausmessung (0,0) für den Pixel am unteren, linken Rand, und (1,1) für den rechten, oberen Pixel [HA01]. Jedem Vertex wird nun eine Koordinate im Texturen-Koordinatensystem zugeordnet. Die (s,t)-Koordinaten müssen ebenfalls wieder in einem *direct FloatBuffer* übergeben werden. Dadurch lässt sich eine Textur gezielt auf eine Oberfläche platzieren. Je nach Wahl der (s,t)-Koordinaten der Vertices, lässt sich die Textur als Ganzes auf eine Oberfläche mappen, nur zu einem Teilbereich oder aber auch mehrmals in wiederholender Reihenfolge. Dafür muss zunächst ein Bild an eine Textur gebunden werden. Dazu generiert OpenGL eine Textur mit einer dazugehörigen ID, die zwischengespeichert wird. Anschließend wird OpenGL in den Zustand zur Bindung einer Textur mit gewünschter ID gesetzt. Daraufhin kann eine Bitmap auf die Textur gebunden werden.

Listing 2.2: Texturgenerierung

```
1 int[] TextureIDs = new int[1];
2 gl.glGenTextures(1, TextureIDs, 0);
3 TextureID = TextureIDs[0];
4 gl.glBindTexture( GL10.GL_TEXTURE_2D, TexturID );
5 GLUtils.texImage2D( GL10.GL_TEXTURE_2D, 0, bitmap, 0);
```

Zuletzt muss die Textur noch konfiguriert werden. Mit der Methode aus Listing 2.3 werden die Parameter für die Art der Texturierung spezifiziert. Dabei können verschieden Methoden für die Konfiguration gewählt werden.

Listing 2.3: Methode zur Texturkonfiguration [Ope12]

```
1 void glTexParameterf(GLenum target,
2 GLenum pname, GLfloat param);
```

target bestimmt die Art der Oberfläche, die texturiert wird. Hier gibt es die Optionen *GL_TEXTURE_2D* und *GL_TEXTURE_CUBE_MAP*. Da in dieser Arbeit lediglich zweidimensionale Oberflächen texturiert werden, wird sich hier auf *GL_TEXTURE_2D* beschränkt.
Um die Textur in gewünschter Weise zu Mappen, muss man verschiedene Einstellungen vornehmen. *pname* wählt die Einstellung aus und *param* setzt das zugehörige, gewünschte Attribut. Für *pname* stehen *GL_TEXTURE_MIN_FILTER*, *GL_TEXTURE_MAG_FILTER*, *GL_TEXTURE_WRAP_S*, und *GL_TEXTURE_WRAP_T* zur Verfügung.
Es soll zunächst der *GL_TEXTURE_MAG_FILTER* betrachtet werden. Um die Funktionsweise des Aufrufes zu verstehen, wird der Begriff *Texel* eingeführt. Ein Texel ist ein Pixel der Textur. Meist sind Anzahl der Pixel der Oberfläche nicht identisch mit der Anzahl der Texel. Der Aufruf mit *GL_TEXTURE_MAG_FILTER* bestimmt die Vorgehensweise in dem Fall, dass die Anzahl der Pixel höher ist, als die Anzahl der Texel. Daraus folgt, dass ein Texel auf mehrere Pixel gemapped werden muss. Mit der Einstellung *GL_NEAREST* wird dem Pixel die Farbe desjenigen Texels gegeben, dessen transformierter Mittelpunkt, dem des Pixels am nächsten liegt. Mit *GL_LINEAR* mittelt erhält das Pixel den gemittelten Farbwert der vier nächstgelegenen Texel.
Analog zu *GL_TEXTURE_MAG_FILTER* bestimmt *GL_TEXTURE_MIN_FILTER* die Vorgehensweise bei niedrigerer Pixelanzahl als der der Texel. Es stehen hier erneut die Optionen *GL_NEAREST* und *GL_LINEAR* zur Verfügung. Des weiteren gibt es die Optionen *GL_NEAREST_MIPMAP_NEAREST*, *GL_NEAREST_MIPMAP_ LINEAR*, *GL_LINEAR_MIPMAP_NEAREST*, *GL_LINEAR_MIPMAP_LINEAR* [Cla97].
Es wurde gezeigt, dass jedem Vertex eine Koordinate im Texturkoordinatensystem (s,t) gegeben wird. Der Fall, dass die Koordinate s > 1(bzw. t > 1) ist, wird mit der Option *GL_TEXTURE_WRAP_S* (bzw. *GL_TEXTURE_WRAP_T*) behandelt. Eine Möglichkeit dies zu lösen ist *GL_CLAMP*. Dieser Aufruf bewirkt, dass die Textur einmalig auf die Oberfläche gemapped wird. Gegebenenfalls wird die Oberfläche nicht komplett von der

Textur ausgefüllt (siehe Abbildung 2.2). Mit dem Aufruf *GL_REPEAT* wird die Textur in wiederholender Reihenfolge auf die Oberfläche gezeichnet (siehe Abbildung 2.3, 2.4).

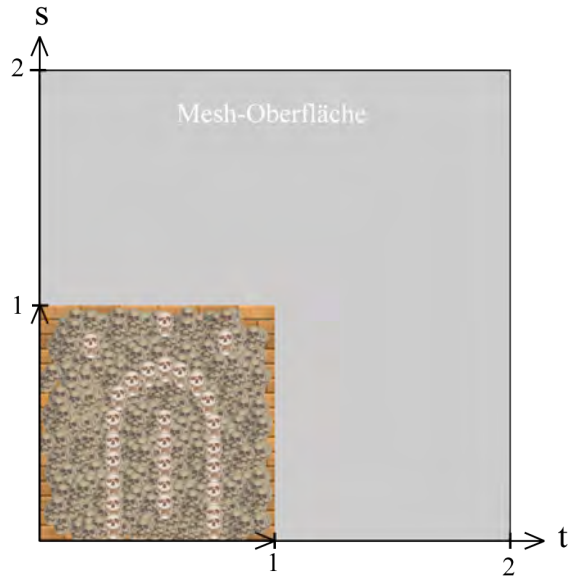

Abbildung 2.2: Textur Mapping mit *GL_CLAMP*

Weitere Möglichkeiten das Überziehen der Oberflächen zu steuern, bieten *GL_CLAMP_TO_EDGE* und *GL_MIRRORED_REPEAT* [Ope12].

2.1.4 Zeichnen mit OpenGL

Im letzten Abschnitt wurde das Mesh erklärt und wie es erstellt wird. Nun soll gezeigt werden, wie OpenGL Objekte einer digitalen dreidimensionalen Welt in ein zweidimensionales Bild umrechnet und diese Informationen in den *FrameBuffer* des Video-RAMs schreibt. Dabei soll stets auf die Zustände nach der *Rendering-Pipeline* eingegangen werden.

2.1.4.1 Model-View Matrix

Die Model-View Matrix verwaltet vereinfacht betrachtet die digitale Welt. Durch das *Model* werden die Positionen der Meshes verwaltet und transformiert. Die Position und

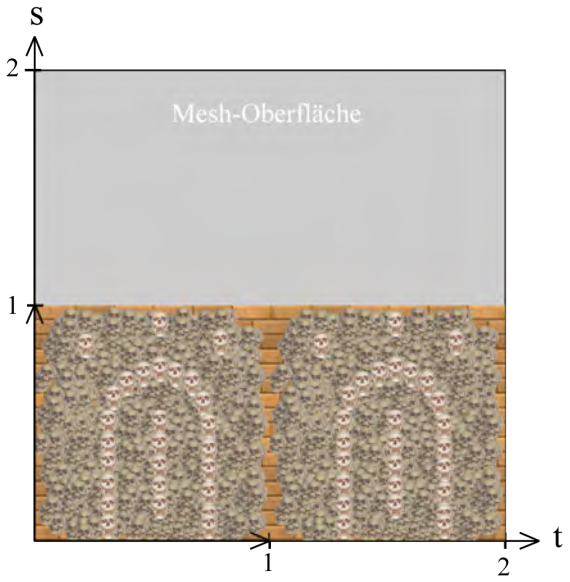

Abbildung 2.3: Textur Mapping mit *GL_CLAMP* in Richtung s und *GL_REPEAT* in Richtung t

der Blickwinkel der Kamera werden durch die *View* bestimmt. Zur Bearbeitung dieser Matrix muss zuvor der Zustand von OpenGL entsprechend gesetzt werden (siehe Listing 2.6).

Listing 2.4: Setzen des Zustands von OpenGL zur MV-Matrix Bearbeitung

```
6 glMatrixMode(GL_MODELVIEW);
```

Model Wie bereits erwähnt, beinhaltet das Model die Meshes. Und wie in dem Abschnitt über Vertices erklärt, sind die Koordinaten der Vertices nur relativ in einem Raum bestimmt. Um die Meshes nun absolut in einem Raum darzustellen, unterstützt OpenGL Matrizenmultiplikationen, die von den Funktionen für Matrizentransformationen aus Listing 2.5 verwendet werden.

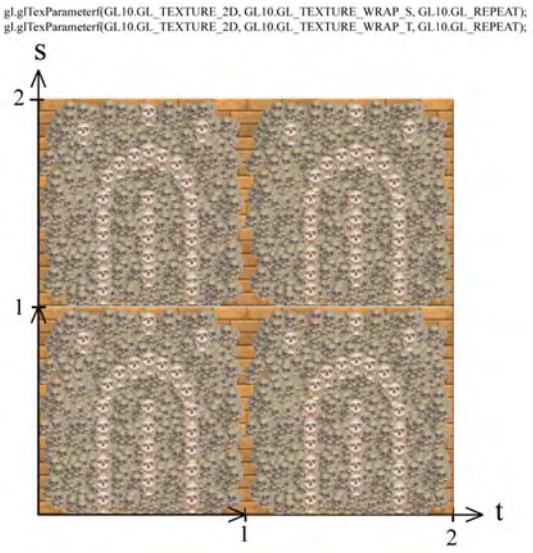

Abbildung 2.4: Textur Mapping mit *GL_REPEAT*

Listing 2.5: Methoden zur Matrixtransformation [Ope12]

```
1 void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
2 void glScalef(GLfloat x, GLfloat y, GLfloat z);
3 void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
```

Die Meshes können mit *glTranslate* im Raum verschoben, mit *glScale* skaliert und mit *glRotate* rotiert werden. *glTranslate* bzw. *glScale* benötigt einen Verschiebungs- bzw. Skalierungsvektor. *glRotate* benötigt eine Drehachse, gegeben als Einheitsvektor sowie den Rotationswinkel. Jede Matrix kann dabei beliebig oft transformiert werden (Siehe Abbildung 2.5). Ebenso kann jede Matrix beliebig viel Objekte beinhalten.

OpenGL arbeitet innerhalb der Model-View Matrix wiederum mit einem Stapel von Matrizen. Dabei wird stets die oberste Matrix verwaltet. Zur Bearbeitung des Stapels dienen die Methoden aus Listing 2.6.

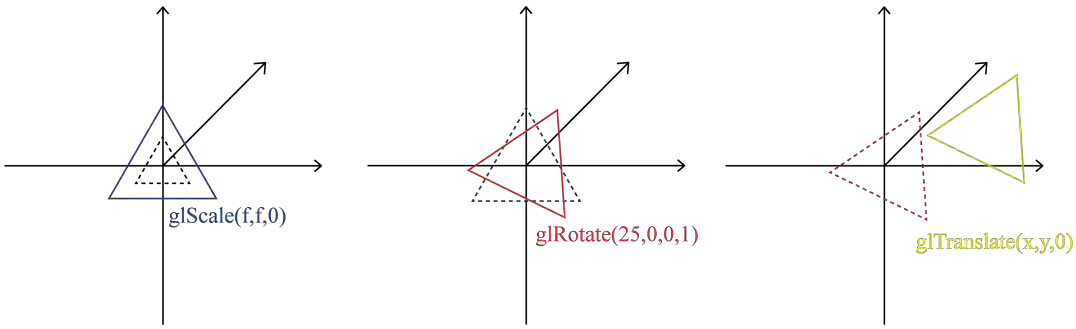

Abbildung 2.5: Transformationen

Listing 2.6: Methoden zur Bearbeitung des Matrizenstacks [Ope12]

```
7 void glLoadIdentity()
8 void glPushMatrix()
9 void glPopMatrix()
```

glLoadIdentity legt die Identitätsmatrix auf den Matrizen Stapel. Die Identitätsmatrix ist eine Matrix, auf der noch keine Transformationen angewandt wurden. Mit *glPushMatrix* wird eine Kopie der aktuellen Matrix auf den Stapel gelegt. *glPopMatrix* entfernt die oberste Matrix von dem Stapel. Nun können mehrere Transformationen auf jeweils verschiedenen Matrizen durchgeführt werden (Abbildung 2.6).

Um ein Mesh abschließend darzustellen, muss das Zeichnen mit Listing 2.7 veranlasst werden.

Listing 2.7: Methode für den Zeichenbefehl [Ope12]

```
1 void glDrawArrays(GLenum mode,GLint first,GLsizei count)
```

Die letzten zwei Parameter der Methode bestimmen den Beginn der Daten im zuvor referenzierten Buffer (*first*) und die Anzahl der Punkte (*count*). Der erste Parameter bestimmt, wie gezeichnet werden soll. In dieser Arbeit werden lediglich die zwei wichtigsten Aufrufe hierfür getätigt. *GL_TRIANGLES* und *GL_TRIANGLE_STRIP*. Der Aufruf mit *GL_TRIANGLES* für *mode* befiehlt OpenGL ein oder mehrere Dreiecke unabhängig

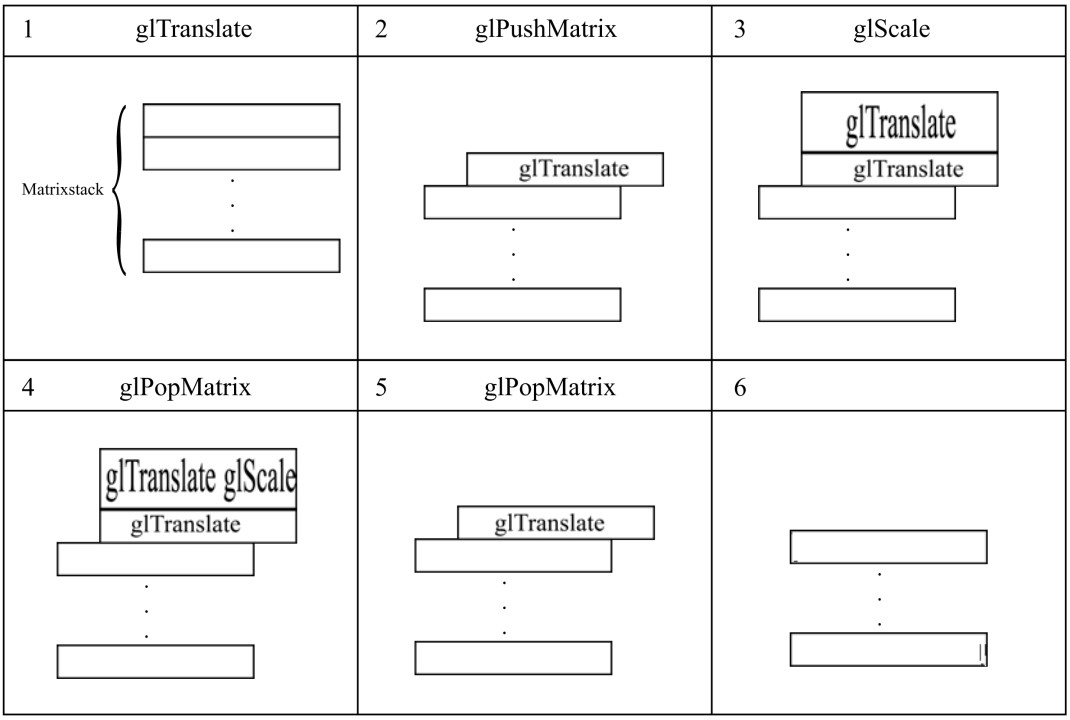

Abbildung 2.6: Matrizenstack bearbeiten

voneinander zu zeichnen. Mit diesem Befehl wird für *count* ein Vielfaches von Drei erwartet, da für jedes Dreieck drei Punkte benötigt werden.
Der Aufruf mit *GL_TRIANGLE_STRIP* ermöglicht mehrere, aneinander gereihte Dreiecke zu zeichnen.

Wie in Abbildung 2.7 gezeigt, werden für das Zeichnen mit *GL_TRIANGLE_STRIP* bei mehreren Dreiecken, statt 3*n Koordinaten nur n+2 benötigt (mit n := Anzahl der zu Zeichnenden Dreiecke). Dreieck A wird mit den ersten drei Koordinaten gezeichnet, Dreieck B mit den Koordinaten 2, 3 und 4 und Dreieck C mit den Koordinaten 3, 4 und 5. Bei weiteren zu zeichnenden Dreiecken würde das letzte Dreieck schließlich mit den Koordinaten (N-2, N-1, N) gezeichnet werden(mit N:= Anzahl Koordinaten).

Der Zeichenbefehl wird aufgerufen, wenn die gewünschte Matrix, die oberste des Matrizenstapel ist. Abbildung 2.8 zeigt das Zeichnen eines auf dem Ursprung liegenden

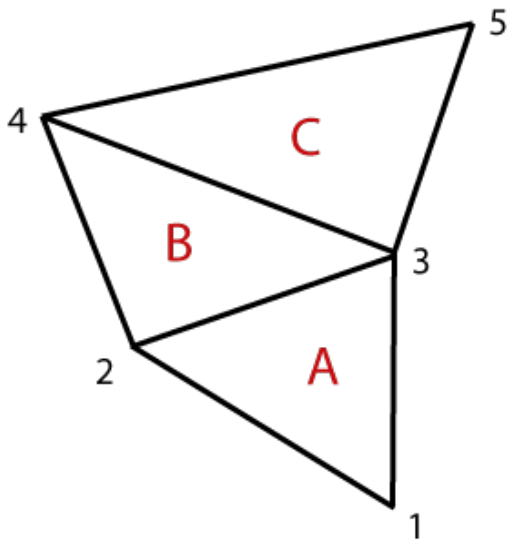

Abbildung 2.7: Drei Dreiecke mit GL_TRIANGLE_STRIP

Dreieckes zwischen Schritt 3 und 4, und ein erneutes Zeichnen zwischen Schritt 4 und 5.

View Die Welt wurde bis jetzt nur aus einem statischen Blickwinkel betrachtet. Die Model-View Matrix erlaubt es sowohl die Position der Kamera als auch den Blickwinkel zu bestimmen. Dazu dient folgende Methode:

Listing 2.8: Methode zum festlegen der Kamera [Ope12]

```
1 gluLookAt(GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,
2 GLdouble centerX, GLdouble centerY, GLdouble centerZ,
3 GLdouble upX, GLdouble upY, GLdouble upZ)
```

Dabei geben die *eye*-Koordinaten (x,y,z) den Blickpunkt an und die *center*-Koordinaten den Punkt der Kamera (Point of View, POV). Mit den *up*-Koordinaten bestimmt man einen Einheitsvektor, der nach oben zeigt. Es ist sinnvoll vor dem *gluLookAt* Aufruf die Einheitsmatrix zu laden.

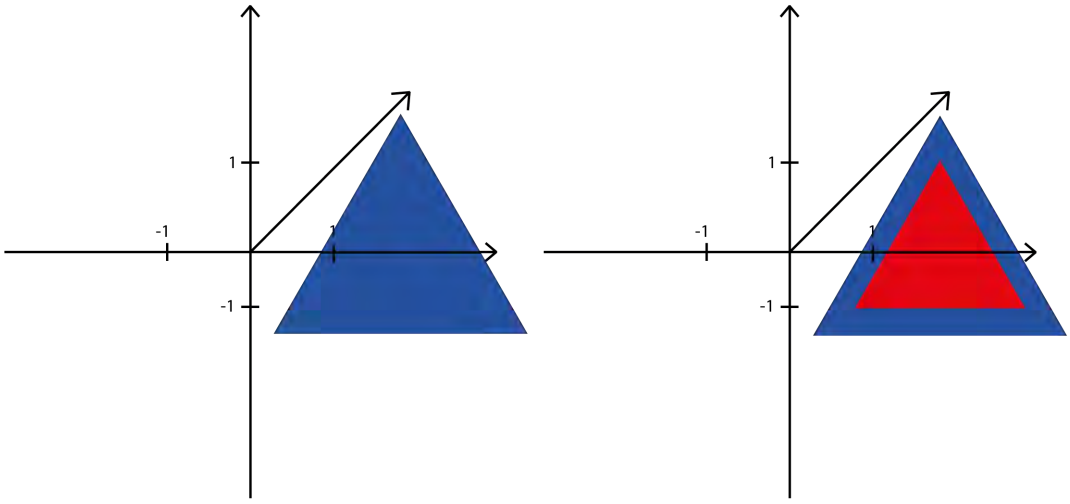

Abbildung 2.8: Zweimaliges Zeichnen desselben Dreieckes in verschiedene Matrizen

2.1.4.2 Projektionsmatrix und View Frustum

Der nächste Schritt ist es, die erstellte Welt auf eine zweidimensionale Fläche zu übertragen. Das ist notwendig, um die Darstellungsinformationen in den FrameBuffer zu laden, der einen zweidimensionalen Bildschirm digital repräsentiert. Man nennt dieses Verfahren *Projektion*.
Mathematisch ausgedrückt ist die Projektion also eine Abbildung $\rho := x_1 \times y_1 \times z_1 \rightarrow x_2 \times y_2$. OpenGL verwaltet im Inneren die dreidimensionale Welt für eine bessere Matrixtransformation, allerdings in einem vierdimensionalen Koordinatensystem und projiziert diese auf eine zweidimensionale Zeichenfläche. Da dies allerdings Berechnungen der höheren Mathematik sind, wird diese Betrachtung in dieser Arbeit vernachlässigt. Um Projektionen zu berechnen, stelle man sich vor, es werden durch jeden Punkt eines Objektes der digitalen Welt Geraden gezogen. Je nach Art der Projektion liegen diese unterschiedlich in einem Raum. Anschließend wird die Schnittmenge der Geraden mit der Projektionsfläche berechnet. Diese Schnittmenge ergibt das Bild im *FrameBuffer*.
Es gibt zwei Arten der Projektion. Die Erste ist die orthogonale Projektion. Bei der *orthogonalen Projektion* verlaufen die Projektionsgeraden parallel zueinander und dem

Namen entsprechend, orthogonal zu der Projektionsfläche. Dabei ist die Entfernung des Punktes zu der Projektionsfläche für die Darstellung irrelevant (siehe Abbildung 2.11). Diese Art der Projektion wird hauptsächlich für zweidimensionale Welten benötigt.
Die andere Weise der Projektion ist die *perspektivische*. Hier sind die Projektionsgeraden, Verbindungen der Objektpunkte mit dem POV. Sie simuliert die Perspektive der Realität. In der Kunst spricht man dabei von der Fluchtpunktperspektive [HA01].

Listing 2.9: Projektion

```
1 gl.glMatrixMode( GL10.GL_PROJECTION );
2 gl.glLoadIdentity();
3 GLU.gluPerspective( gl, 67, aspectRatio, 1, 100 );
```

Um eine Projektion auszuführen, wird OpenGL zuerst in den Zustand zur Bearbeitung der Projektionsmatrix überführt. Danach wird die Einheitsmatrix geladen. Der Aufruf *gluPerspective* benötigt nun fünf Übergabeparameter. Der Erste ist das *GL10*-Interface (siehe Abschnitt 2.1.6). Der zweite Aufruf ist der Winkel des Sichtfeldes in Grad. Dieser beträgt beim Menschen in etwa 67 Grad. Der dritte Parameter, der *aspectRatio*, gibt das Seitenverhältnis der beiden *clipping panes* an. Die letzten beiden Parameter geben die Entfernung der *near* und *far clipping pane* an. Vor der *near clipping pane* sowie nach der *far clipping pane*, wird OpenGL nichts mehr darstellen [Zec11].
OpenGL überschreibt derzeit nach jedem Zeichenaufruf den *FrameBuffer*, unabhängig von den Tiefenpositionen der Objekte. So würde das zuletzt gezeichnete Objekt, ungeachtet der Postionen, vorherige Objekte überdecken. Eine Lösung hierfür bietet OpenGL mit dem Aufruf *glEnable(GL_DEPTH_TEST)*. Dadurch legt OpenGL zusätzlich zum FrameBuffer einen Z-Buffer an, in dem es die Tiefeinformation eines jeden Pixels speichert. So überschreibt OpenGL ein Pixel des *FrameBuffers* nur dann, wenn der neu zu zeichnende Pixel sich näher an der Projektionsfläche befindet, als der vorherige [Zec11].

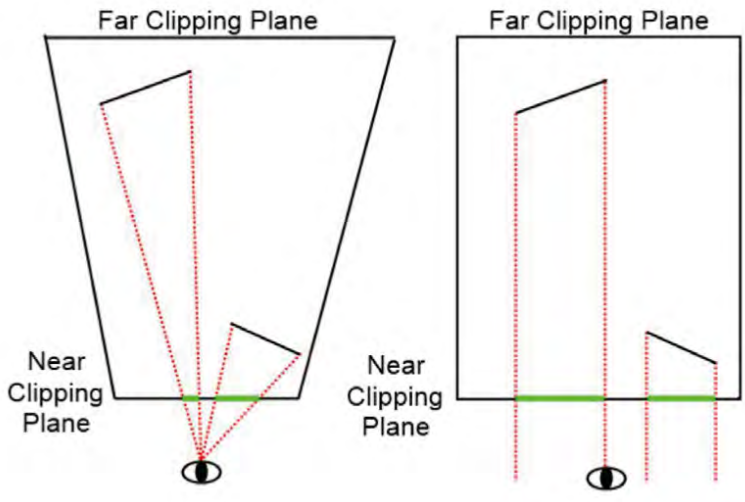

Abbildung 2.9: Orthogonale und perspektivische Projektion

2.1.5 Lichtgestaltung

In diesem Abschnitt soll auf die Lichtgestaltung unter OpenGL eingegangen werden. Da in dieser Arbeit Materialeigenschaften und *Normale* außer Acht gelassen werden, soll hier nur ein grober Überblick gegeben werden. *Normale* sind in der Mathematik Vektoren die orthogonal zu einer Ebene stehen. Sie dienen in der Grafikprogrammierung der Berechnung der Lichtreflexion. Die *Normalen* können variiert werden, um unterschiedliche Reflexionen zu erzeugen.
Es gibt in OpenGL vier verschiedene Optionen eine Szene zu beleuchten [Zec11]:

1. *Direktionales Licht* hat keine konkrete Lichtquelle. Die “Lichtwellen“ verlaufen alle parallel zu einem gegebenen Einheitsvektor. Das ist mit dem Lichteinfall der Sonne zu vergleichen.

2. *Punktlicht* ist ein Licht, ausgehend von einem einzigen Punkt. Die “Lichtwellen“ verlaufen gleichmäßig in alle Richtungen. Vergleichbar mit dem Licht einer Straßenlaterne.

3. *Spotlight* ist ein Scheinwerferlicht, das von einem bestimmten Punkt ausgeht und in eine bestimmte Richtung, mit einem bestimmten Winkel scheint.

4. *Ambientes Licht* bestimmt die Helligkeit, die von unbeleuchteten Objekten ausgestrahlt wird.

Dazu gibt es verschiedene Arten von Licht, die bestimmen wie das Licht von Objekten emittiert wird (*ambientes*, *diffuses*, *spektakuläres* und *emmissives* Licht) [HA01]. Damit lassen sich Materialeigenschaften definieren. Für jede Oberfläche kann reguliert werden, welches Licht wie stark reflektiert werden soll. Somit lassen sich Materialeigenschaften bezüglich der Lichteffekte simulieren. OpenGL ermöglicht es, dies mit *glEnable(GL10.GL_COLOR_MATERIAL)* zu automatisieren.
OpenGL bietet bis zu acht verschiedene Lichtquellen an. Für jedes verwendete Licht, muss die Art der Lichtquelle (ambientes Licht, Punktlicht, direktionales oder Scheinwerferlicht) bestimmt und der RGBA-Farbraum für jeden Lichttyp (ambientes, diffuses, spektakulär) definiert werden (emmisives Licht ist hier ein Spezialfall und kann vernachlässigt werden). Punktlicht benötigt zusätzlich noch Angaben zur Position und Spotlicht Positions- und Richtungsinformationen (Einheitsvektor) sowie Angaben zum Ausfallwinkel des Lichtes.

2.1.6 OpenGL unter Android

Damit OpenGL für Android-Anwendungen verwendet werden kann stellt Android die *GLSurfaceView*, das *Renderer*-Interface sowie mehrere Schnittstellen zu OpenGL zur Verfügung (Vgl. Abbildung2.11). In dieser Arbeit wird die Schnittstelle *GL10* verwendet, die die Funktionen der OpenGL 1.0 Spezifikationen beinhaltet.

Zunächst wird ein Fenster benötigt, in dem OpenGL zeichnen kann. Dieses wird mit der *GLSurfaceView* zur Verfügung gestellt. Diese *View* erbt von der *SurfaceView* und wird analog zu gewöhnlichen *Views* in eine *Activity* eingebunden. Zusätzlich verwaltet die *GLSurfaceView* das *Renderer*-Interface, welches von der Anwendung implementiert wird. Im *Renderer*-Interface werden die OpenGL-Befehle aufgerufen und ist der Kern des späteren Spiels. Das *Renderer*-Interface implementiert drei Methoden (Vgl. Listing 2.10).

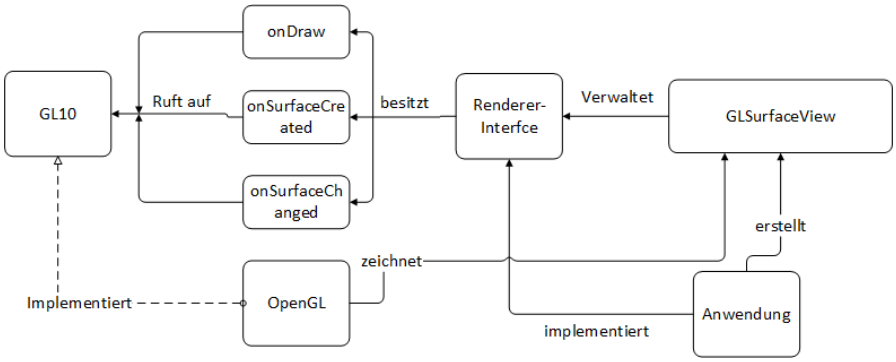

Abbildung 2.10: Zusammenhang zwischen OpenGL und Android

Listing 2.10: Methoden des Renderer-Interfaces [Inc14]

```
1 public void onSurfaceCreated(GL10 gl, EGLConfig config)
2 public void onSurfaceChanged(GL10 gl, int width, int height)
3 public void onDraw(GL10 gl)
```

Dabei wird das Interface *GL10* übergeben. Über das *GL10* können die OpenGL Funktionen aufgerufen werden, welche anschließend vom Grafikkartenchip ausgeführt werden. *onSurfaceCreated* wird aufgerufen, sobald eine *Activity* eine *GLSurfaceView* erstellt. Hier können alle Objekte erstellt und Texturen initialisiert werden, die für das spätere Zeichnen benötigen werden. Dadurch wird der *Setup* des späteren Spieles realisiert.*onSurfaceChanged* wird aufgerufen, sollte sich die Größe des *Views* ändern. In *onDraw* wird der *Game-Loop* verwirklicht. *onDraw* wird iterativ, sooft und schnell wie möglich aufgerufen. Hier wird das Zeichnen veranlasst, Meshes transformiert und alle weiteren Veränderungen während der Laufzeit des Spieles realisiert [Zec11].

2.2 Labyrinth

Ein Labyrinth ist ein Gebilde, das einen Anfang und ein Ziel besitzt und mit unterschiedlich verzweigten Gängen das Auffinden des Zielweges zu einem Rätsel macht. Das Ziel muss dabei durch Ablaufen der Gänge gefunden werden. Der Algorithmus zur Generierung des Labyrinthes bildet ein Labyrinth als mathematischer Graph (V,E) ab. Man setze

dazu in einem Labyrinth nun Wegpunkte und verbindet diese so, das jeder Wegpunkt mindestens eine Verbindung besitzt. Jede Verbindung ist nun ein Weg in dem Labyrinth. Dort wo keine Verbindungen sind, stehen Wände. Dabei ist die Menge der Wegpunkte bestimmt durch die Menge der Knoten (V) und die Anzahl der Verbindungen ist gegeben durch die Menge der Kanten (E). Man unterscheidet dabei zwischen verschiedene Arten von Labyrinthen indem man verschiedene Tesselationen betrachtet. Tesselation bedeutet die Aufteilung des Raumes in geometrische Grundformen.

Othogonale Tesselation In dieser Arbeit werden Labyrinthe mit orthogonaler Tesselation verwendet. Unter orthogonaler Tesselation versteht man, dass die Knoten des Graphen orthogonal zueinander angeordnet sind. Dabei kann ein Knoten maximal vier Verbindungen haben.

Theta Tesselation Bei der Theta Tesselation sind die Knoten des Labyrinthes Ringförmig angelegt. Dabei ist entweder Ziel- oder Startpunkt in der Mitte des Ringes. Alle weiteren Knoten sind auf verschieden Große Kreise mit dem selben Mittelpunkt angeordnet.

Formlose Tesselation Hier sind Knoten und Verbindungen willkürlich angeordnet und verteilt.

Zeta Tesselation Zeta Tesselation sind analog zur orthogonalen Tesselation angeordnet, erlauben jedoch 45 Grad Verbindungen.

[Fol08]

Generierung

Der Algorithmus zur Generierung eines Labyrinthes bedient sich dem Prinzip der Tiefensuche. Dabei liegt das Labyrinth mit gewünschten Ausmaßen als Gitternetz vor. Zu Beginn gibt es noch keine Verbindungen. Der Algorithmus beginnt an einem beliebigen

Punkt und markiert diesen als besucht. Dann wird ein beliebiger, noch nicht besuchter, Nachbarknoten besucht und eine Verbindung zu diesem erstellt. Dies geschieht solange bis kein unbesuchter Nachbarknoten mehr existiert. Via Backtracking wird solange der bereits erstellte Weg zurückgegangen bis ein Knoten mit einem unbesuchten Nachbarn gefunden wurde. Vorherige Schritte werden anschließend von diesem Knoten aus durchgeführt. Dies geschieht iterativ bis jeder Knoten besucht wurde [Tur09].

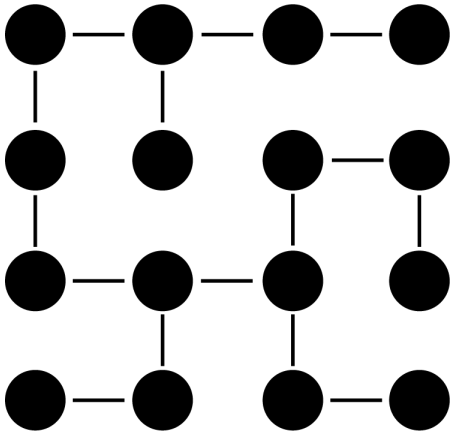

Abbildung 2.11: Graphische Darstellung eines mit der Tiefensuche erstellten Labyrinthes mit orthogonaler Tesselation

2.3 Tinnitus

Ein Tinnitus (lat. Klingeln) bezeichnet die Wahrnehmung eines Pfeiftones, für den keine physikalische Ursache in der näheren Umgebung existiert. Dabei unterscheidet man zwischen einem subjektiven und objektiven Tinnitus. Während dem objektiven Tinnitus eine organische Fehlfunktion vorliegt und von einem Arzt gemessen werden kann, ist die Ursache für einen subjektiven Tinnitus ein Hörsturz oder eine chronische Überbelastung und hat einen psychischen Ursprung [SRLP$^+$13]. Da die Tinnituswahrnehmung sehr individuelle Ursachen hat, ist der Tinnitus durch keine Standardtherapie behandelbar [Lin14, SHP$^+$14b]. Um die individuellen Schwankungen der Tinnituswahrnehmung besser zu erfassen, wurde in einem Forschungsprojekt die mobile Applikation *Track Your Tinnitus* entwickelt. Diese App erfasst mittels verschiedener Fragebögen Schwankungen

und Grad des Tinnitus und wie diese mit dem Tagesablauf und Aktivitäten des Patienten zusammenhängen [Ini13, SSP+14, SHP+14a]. Das Problem, dass bei einer Befragung eines Tinnitus-Patienten häufig auftritt, ist dass der Patient sich auf seinen Tinnitus konzentrieren muss und ihn anschließend als stärker wahrnimmt als zuvor. Dieses Problem soll nach der Befragung gelindert werden, indem neue kognitive Reize den Patienten von seinem Tinnitus ablenken. Dazu soll das in dieser Arbeit entwickelte Spiel die *Track Your Tinnitus*-App so erweitern, dass die Konzentration des Benutzers auf die räumliche Wahrnehmung und Orientierung liegt.

3

Anforderungen

In dieser Arbeit soll ein begehbares Labyrinth in einer dreidimensionalen Welt konzipiert und implementiert werden und für zwei Anwendungsfälle spezifiziert werden. In diesem Kapitel werden die Anforderungen erläutert und in funktionale Anforderungen (FA) und nichtfunktionale Anforderungen (NFA) kategorisiert. Die allgemeinen Anforderungen an das Labyrinth werden in Abschnitt 3.1 formuliert. Die für das Labyrinth als Teil der *Track Your Tinnitus*-App bestehenden Anforderungen werden in Abschnitt 3.2 erläutert. Außerdem werden die Anforderungen, die für ein selbständiges Spiel benötigt werden, in Abschnitt 3.3 beschrieben.

3.1 Anforderungen an das Labyrinth

Bevor die Anforderungen an die spezifischen Anwendungsfälle erörtert werden, werden die Anforderungen an das zugrunde liegende Spiel betrachtet.

FA #1 Labyrinth Es soll eine Labyrinthstruktur mit orthogonaler Tesselation generiert werden, auf der das Spiel aufbaut.

FA #2 Swipegesten *Swipegesten* sollen eine Bewegung auf einem Gitternetz ermöglichen da die Wischbewegungen konform der Spielbewegung sind und die Steuerung daher intuitiver ist.

FA #3 Schwierigkeitsgrad Der Schwierigkeitsgrad soll veränderlich sein damit die Level den Fähigkeiten des Spielers angepasst werden können.

FA #4 Levelaufbau Damit die Level nicht einzeln erstellt werden müssen, sollen die Levels automatisch generiert werden. Dadurch wird die Anzahl der Level, nur durch die Anzahl der Verbindungsmöglichkeiten der Knoten des Graphen des Labyrinthes begrenzt. Außerdem wird dadurch der Speicher gespart, der nötig ist verschiedene Level fest zu gestalten.

FA #5 Licht Das Spiel soll eine Lichtgestaltung realisieren um die Atmosphäre authentischer wirken zu lassen.

FA #6 Items Sollte ein Spieler Probleme bei der Wegfindung haben, sollen ihm Items dabei behilflich sein.

FA #7 Score Für den Anreiz das Labyrinth möglichst gut zu absolvieren, soll ein Score den Spieler ein Feedback über dessen Leistung geben. Die Güte des Abschlusses soll sich aus der benötigten Zeit sowie den benötigten Schritten definieren.

FA #8 Informationen Damit der Spieler während des Spieles Informationen zu dem laufendem Spiel erhält, wie seine *Credit*-Anzahl, begangene Schritte und dem aktuellen *Rating*, soll eine Informationsleiste implementiert werden.

FA #9 Spielwährung Eine spielinterne Währung soll es ermöglichen, Items zu erwerben damit der Spieler Hilfe bei der Wegfindung erhält.

FA #10 Map Es soll eine Karte aufrufbar sein, die der Spieler beim Finden des Zielweges unterstützt.

FA #11 Kompass Ein soll ein Kompass erworben werden können, der dem Spieler zur Orientierungshilfe dient.

FA #12 Zielpfeil Es soll ein Zielpfeil erworben werden können, der dem Spieler bei der Wegwahl Entscheidungshilfe bietet.

NFA #1 Rechteckiges Labyrinth Damit der Schwierigkeitsgrad des Spieles nicht zu komplex wird und es einfach darzustellen ist, soll das Labyrinth eine zweidimensionale Struktur besitzen. Zusätzlich soll der Spieler stets einschätzen können, welche Wege die einzelnen Abzweigungen ermöglichen. Dafür soll die Form des Labyrinthes immer ein Rechteck sein.

NFA #2 Vollständiges Labyrinth Eine weitere Voraussetzung dafür, dass der Spieler stets einschätzen kann, welche Möglichkeiten bestimmte Abzweigungen bieten, ist, dass jeder Punkt auf dem Rechteck erreichbar ist.

NFA #3 Zielweg Damit der Spieler das Ziel möglichst nicht zufällig findet, soll es nur einen einzigen Weg zum Ziel geben.

NFA #4 Spielumgebung Die zweidimensionale Labyrinthstruktur soll in einer dreidimensionalen Umgebung implementiert werden. Dadurch soll der Rätselfaktor erhöht werden, da der Zielweg nicht bekannt ist und durch Erkundung entdeckt werden muss. Zudem wird somit die räumliche Orientierungsfähigkeit des Spielers gefordert. Zusätzlich soll dadurch die Illusion entstehen, man befände sich in einem echten Labyrinth.

NFA #5 Einfache Bedienung Die Bewegung durch das Labyrinth soll durch einfache Bedienung leicht und schnell steuerbar, sein damit der Spieler sich bestens auf die Wegfindung konzentrieren kann.

NFA #6 Levelart Es soll verschieden große und unterschiedlich komplexe Level realisiert werden damit der Schwierigkeitsgrad des Labyrinthes angepasst werden kann.

NFA #7 Atmosphäre Damit die Illusion, man befände sich in einem echten Labyrinth möglichst effektiv wird, soll durch natürlicher Lichtgestaltung und kontextbezogenen Texturen eine authentische Atmosphäre entstehen. Ebenso benötigt es dafür eine abwechslungsreiche Spielewelt.

NFA #8 Texturen Die Texturen des Spiels sollen im Kontext einer ägyptischen Pyramide gewählt werden damit das Labyrinth eine nachvollziehbare Umgebung besitzt.

NFA #9 Labyrinthgestaltung Durch möglichst vielen verschiedenen Texturen soll eine abwechslungsreiche Welt entstehen.

NFA #10 Performance Es soll auf eine gute Performance geachtet werden. Das heißt, dass die Zeit für die Levelerstellung, sowie die benötigte Rechenleistung für das Zeichnen der Welt minimiert werden soll.

3.2 Anforderungen im Kontext “Track Your Tinnitus“

Das Ziel des Spieles im Kontext “TRACK YOUR TINNITUS“ ist es den Tinnitus-Patienten nach Abschluss des Fragebogens von seinem Tinnitus abzulenken. Dafür bedarf es einiger Anforderungen.

FA #13 Aufmerksamkeit Das Spiel soll möglichst die volle Aufmerksamkeit des Spielers erfordern, um den Spieler von seinen Tinnitus abzulenken.

FA #14 Schwierigkeitsgrad Der Schwierigkeitsgrad des Labyrinthes soll sich automatisch an den Fähigkeiten des Spielers anpassen damit dieser sich stets auf das Spiel konzentrieren muss und somit von seinem Tinnitus abgelenkt wird.

FA #15 Scorekategorisierung Bei einem guten Abschluss soll der Schwierigkeitsgrad erhöht, bei schlechtem Abschluss verringert und bei mittleren Abschluss konstant gehalten werden. Dafür muss der berechnete Score kategorisiert werden, um den Schwiergigkeitsgrad anpassen zu können.

FA #16 Bestenliste Um zusätzlich die Motivation zu erhöhen einen möglichst guten *Score* zu erhalten und somit die Konzentration des Spielers, soll es eine Online-Bestenliste geben, in der man sich mit anderen Spielern und Freunden vergleichen kann.

FA #17 Freundesliste Damit man sich mit seinen Freunden messen kann soll eine Freundesliste erstellt und verwaltet werden können.

FA #18 Server Es wird eine Server Anbindung benötigt, die Daten zu den Benutzern erfasst und zur Verfügung stellt. Der Server benötigt eine Datenbank, die Daten zu den Benutzern, den Scores der einzelnen Level, sowie eine Freundesliste verwaltet.

NFA #11 Dauer Da das Spiel nur eine Erweiterung der *Track Your Tinnitus*-App ist, soll das Labyrinth in etwa 15 Sekunden gelöst sein. Um das zu gewährleisten, soll die benötigte Zeit in den Score mit einfließen.

3.3 Anforderungen an das eigenständige Spiel

Im Vordergrund eines eigenständiges Spieles ist ein möglichst hoher und andauernder Spielspaß. Dafür werden eigenständige Anforderungen benötigt.

FA #18 Schwierigkeitsgrad Der Schwierigkeitsgrad des Labyrinthes soll sich stetig und beständig steigern damit das Spiel fordernd bleibt.

NFA #12 Einsteigerfreundlichkeit Das Spiel soll einsteigerfreundlich sein, damit der Spieler von Anfang an Erfolgserlebnisse erhält, die ihn zum Weiterspielen ermutigen. Deshalb soll zu Beginn eine einfache Labyrinthstruktur gegeben sein.

NFA #13 Langzeitmotivation Damit das Spiel auch auf längere Distanz Spielspaß bringt, soll es eine hohe Langzeitmotivation bieten.

NFA #14 Levelanzahl Die Anzahl der Level soll nicht beschränkt sein um eine möglichst hohe Langzeitmotivation zu erhalten.

3.4 Zusammenfassung

Hier werden die abgefassten Anforderungen nochmals tabellarisch dargestellt.

Tabelle 3.1: Allgemeine Anforderungen

Kategorie	**Anforderung**	**Beschreibung**
FA #1	Labyrinth	Es soll eine Labyrinthstruktur mit orthogonaler Tesselation generiert werden.
FA #2	Swipegesten	Der Bewegungsaparat soll durch Swipegesten bedient werden.
FA #3	Schwierigkeitsgrad	Der Schwierigkeitsgrad soll anpassbar sein.
FA #4	Levelaufbau	Die Level sollen automatisch erstellt werden.
FA #5	Licht	Die Welt soll in einer authentischen Lichtgestaltung erscheinen.
FA #6	Items	Items sollen erworben werden können, die den Spielern bei der Wegfindung unterstützen.
FA #7	Score	Es soll einen Score bezogen auf die Leistung des Spielers geben.
FA #8	Informationen	Der Spieler soll Informationen zum laufendem Spiel erhalten.
FA #9	Spielwährung	Es soll eine Spielwährung existieren, mit der Items erworben werden können.
FA #10	Map	Als Item soll eine Karte des Labyrinthes existieren.
FA #11	Kompass	Es soll ein Kompass zur Verfügung stehen.
FA #12	Zielpfeil	Ein Zielpfeil, der stets in Richtung Ziel zeigt, soll erworben werden können.

Kategorie	Anforderung	Beschreibung
NFA #1	Rechteckiges Labyrinth	Die Grundform des Labyrinthes soll ein Rechteck sein.
NFA #2	Vollständiges Labyrinth	Jeder Punkt des Labyrinthes soll begehbar und erreichbar sein.
NFA #3	Zielweg	Es soll nur einen einzigen Zielweg geben.
NFA #4	Spielumgebung	Das Spiel soll in einer dreidimensionalen Umgebung stattfinden.
NFA #5	Bedienung	Die Bewegungen sollen einfach und schnell zu steuern sein.
NFA #6	Levelart	Es soll verschieden große und unterschiedlich komplexe Level geben.
NFA #7	Atmosphäre	Es soll eine authentische Atmosphäre geschaffen werden.
NFA #8	Texturen	Die Texturen soll eine ägyptische Pyramide darstellen.
NFA #9	Labyrinthgestaltung	Es soll eine abwechslungsreiche Welt entstehen.
NFA #10	Performance	Das Spiel soll eine gute Performance aufweisen.

Tabelle 3.2: Anforderungen im Kontext *Track Your Tinnitus*

Kategorie	**Anforderung**	**Beschreibung**
FA #13	Aufmerksamkeit	Das Labyrinth soll möglichst die volle Aufmerksamkeit des Spielers erfordern.
FA #14	Schwierigkeitsgrad	Der Schwierigkeitsgrad soll sich automatisch den Fähigkeiten des Spielers anpassen.
FA #15	Score-kategorisierung	Der errechnete Score soll in “gut’", “mittel“ und “schlecht“ Kategorisiert werden.
FA #16	Bestenliste	Eine Bestenliste soll den Spielern einen Vergleich mit anderen Spielern und Freunden ermöglichen.
FA #17	Freundesliste	Es soll eine Freundesliste verwaltet werden können.
FA #18	Server	Es soll eine Serveranbindung zur Verwaltung der Daten existieren.
NFA #11	Spieldauer	Die Spieldauer soll circa 15 Sekunden betragen.

Tabelle 3.3: Anforderungen an ein eigenständiges Spiel

Kategorie	**Anforderung**	**Beschreibung**
FA #18	Schwierigkeitsgrad	Der Schwierigkeitsgrad soll sich mit höheren Levels stetig steigern.
NFA #12	Einsteigerfreund-lichkeit	Das Labyrinth soll von Beginn an leicht zu erlernen sein und nicht zu schwer sein.
NFA #13	Langzeitmotivation	Das Spiel soll auch über längere Zeit motivieren.
NFA #14	Levelanzahl	Die Anzahl der Level soll nicht beschränkt sein.

4

Implementierung

In diesem Kapitel soll die Konzeption und die Entwicklung des Spieles gezeigt werden und wie es für die jeweiligen Anwendungsfälle optimiert wurde. Die Basis der Anwendungsfälle stellt das Kernspiel dar. Daher wird dessen Implementierung zuerst behandelt. Es folgen die Implementierungen für den Kontext *Track Your Tinnitus* und abschließend die Entwicklung des eigenständigen Spieles.

4.1 Grundlagen zum Labyrinth

In diesem Abschnitt soll die Realisierung der definierten Anforderungen an das Kernspiel gezeigt werden. Dazu wird zunächst die Architektur und der Aufbau des Kernspiels veranschaulicht. Anhand der Architektur wird anschließend der Aufbau der einzelnen Komponenten gezeigt.

4.1.1 Architektur

Um das Spiel zu realisieren muss ein *Game Prozess* das dynamische Spiel erstellen, verwalten und die Schnittstelle zu OpenGL und Android ausfüllen. Abbildung 4.1 zeigt die Eingliederung des *Game Prozesses* in den in Kapitel 2 erläuterten Zusammenhang von Android und OpenGL.

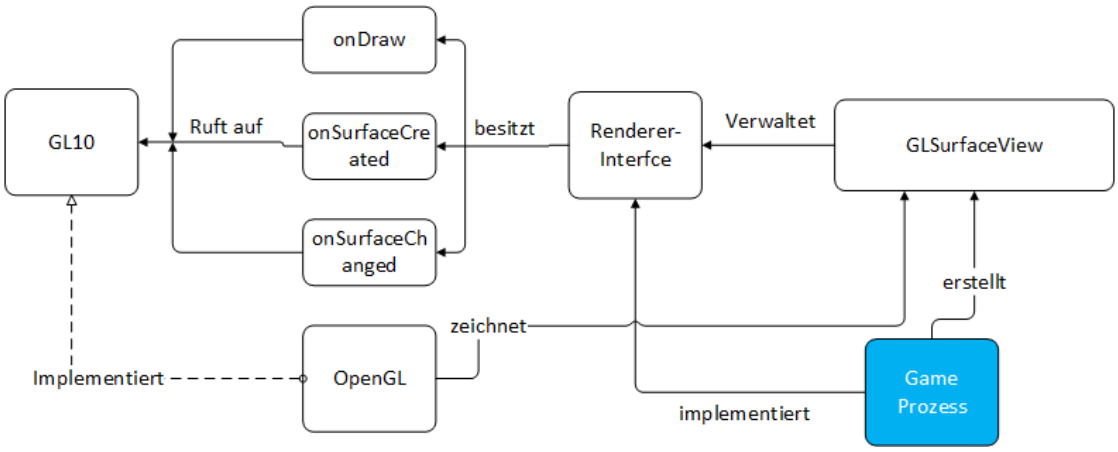

Abbildung 4.1: Eingliederung des Game Prozesses in den OpenGL Prozess

Dabei ist zu erkennen, dass im *Game Prozess* das *Renderer*-Interface sowie die *GL-SurfaceView* erstellt und implementiert werden muss. Den Grundstein dazu legt der *Game Controller*, die Klasse *GameCtrl*. In ihr wird die *GLSurfaceView* initialisiert und die Methoden des *Renderer*-Interface implementiert. Zudem werden im *Game Controller* die Benutzereingaben entgegengenommen und verarbeitet. Um dies vom restlichen Spiel zu trennen, erbt die Klasse *GameFragment* von der Klasse *GameCtrl* und kommuniziert mit den Methoden des *Renderer*-Interfaces über den*GameListener* (siehe Abbildung 4.2).

Ausgehend vom *GameFragment* kann nun der *GameLoop* realisiert werden, der die Schaltzentrale des Spieles darstellt und die einzelnen Klassen miteinander koordiniert (siehe Abbildung 4.3).

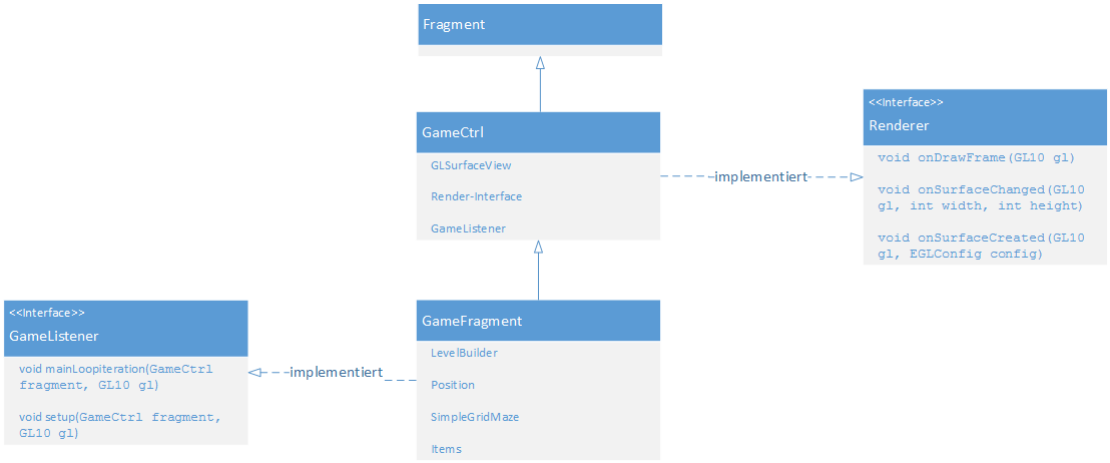

Abbildung 4.2: Klassendiagramm zum *GameCtrl*

4.1.2 Game Controller

Die Klasse *Game Controller* ist die zu Grunde liegende Klasse des *GameFragment*. Sie ist eine abstrakte Klasse, die für eine übersichtlichere Gliederung sorgt. Der *Game Controller* erbt von der von Android zur Verfügung gestellten Klasse *Fragment*, die ein Fenstersystem ähnlich einer *Activity* erlaubt. Hier wird die *GLSurfaceView* verwaltet und in das Fenstersystem eingebunden. Des Weiteren wird im *Game Controller* das *Renderer*-Interface erstellt, implementiert und dem *GLSurfaceView* für spätere Aufrufe übergeben. Zur besseren Übersicht trennt man den *GameLoop* und das Spiel-Setup vom *Game Controller*, indem man sich der Schnittstelle *GameListener* bedient (siehe Listing 4.1). Dies wird dadurch realisiert, dass man in der *onSurfaceCreated*-Methode des *Renderer*-Interfaces die Methode *setup* des *GameListeners* aufruft. Analog wird die *MainLoopIteration* in der *onDraw* Methode aufgerufen. Allerdings wir hier zusätzlich noch das Zeitintervall zwischen den iterativen Aufrufen berechnet. Die sogenannte *DeltaTime*. Diese wird später benötigt, um eine flüssige Bewegung zu simulieren (Vgl. Abschnitt 4.1.3.3).

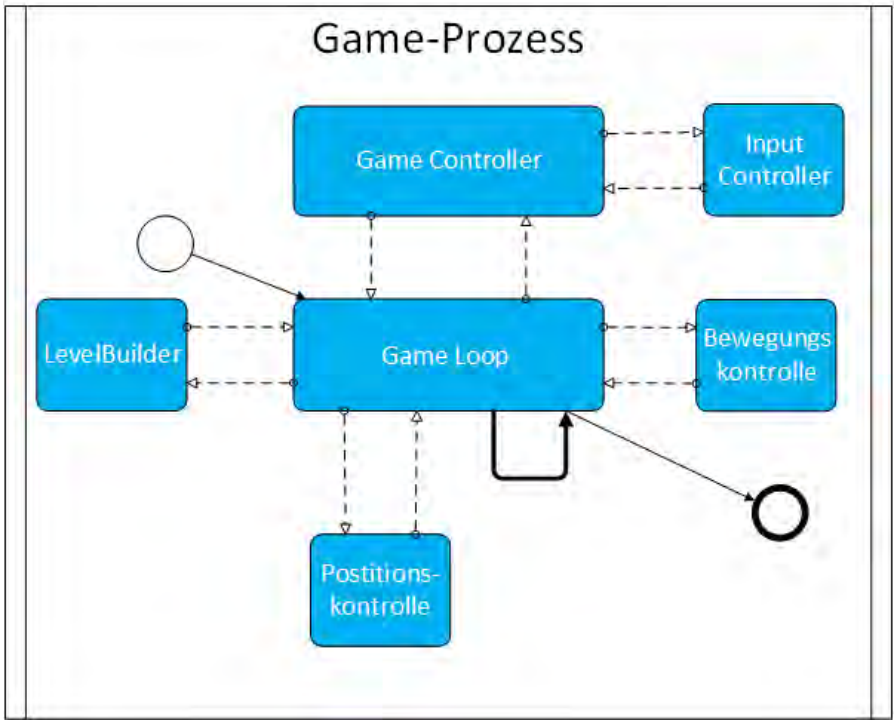

Abbildung 4.3: GameProzess

Listing 4.1: Implemtierung der Renderer-Methoden

```
1
2  @Override
3  public void onDrawFrame(GL10 gl) {
4          long currentDraw=System.nanoTime();
5      deltaTime = (currentDraw-lastDraw) / 1000000000.0f;
6      lastDraw = currentDraw;
7      if(gameListener!=null){
8          gameListener.mainLoopIteration(this, gl);
9      }
10 }
11 @Override
12 public void onSurfaceChanged(GL10 gl, int width, int height) {
13         this.viewportWidth=width;
```

```
14          this.viewportHeight=height;
15          }
16 @Override
17 public void onSurfaceCreated(GL10 gl, EGLConfig config) {
18          if(gameListener!=null){
19                  gameListener.setup(this, gl);
20          }
21 }
```

Zusätzlich verwaltet der *Game Controller* die Benutzereingaben mittels eines *OnTouch-Listeners*. Damit die Swipegesten realisiert werden können, wird die Bewegungsspanne der Berührungspunkte berechnet. Dazu werden alle weiteren Berührungspunkte mit dem Erstberührungspunkt verglichen (Vgl. Listing 4.2. Da der *GameLoop* threadbasiert arbeitet, werden in dem *Game Controller* die Berührungspunkte sowie die Bewegungsspanne für die Abfragen der Bewegungskontrolle gespeichert.

Listing 4.2: Verwaltung der Berührungspunkte

```
1  if( event.getAction() == MotionEvent.ACTION_DOWN ){
2          firstTouchX=(int) event.getX();
3          firstTouchY=(int) event.getY();
4          }
5  if( event.getAction() == MotionEvent.ACTION_DOWN ||
6  event.getAction() == MotionEvent.ACTION_MOVE )
7          {
8          touchX = (int)event.getX();
9          touchY = (int)event.getY();
10         movedX=firstTouchX-event.getX();
11         movedY=firstTouchY-event.getY();
12         isTouched = true;
13         firstTouchX=event.getX();
14         firstTouchY=event.getY();
```

```
15          }
16 if( event.getAction() == MotionEvent.ACTION_UP ){
17          isTouched = false;
18
19          movedX=0;
20          movedY=0;
21 }
```

4.1.3 GameLoop

Der wohl wichtigste Teil der Software stellt der *GameLoop* dar. In Kapitel 2 wurde erläutert, dass der *GameLoop* der iterative Aufruf der *onDraw*-Methode des *Renderer*-Interfaces ist. Allerdings geschieht in dem *GameLoop* (Oder: *MainLoopIteration*) mehr als das reine Zeichnen der Welt.
Zunächst wird ein *Fragment* benötigt, das von *GameCtrl* erbt. Das *GameFragment*. In diesem wird nun der *GameListener* implementiert. Nun können die Methoden des *GameListeners* überschrieben werden. Im *setup*-Aufruf wird alles erstellt, das vor dem ersten *GameLoop* benötigt wird. Es wird das Labyrinth erstellt, der Levelaufbau angestoßen, Ziel und Anfangspunkt berechnet und die Zeit für die Scoreberechnung beginnt zu zählen. Als nächstes wird die *mainLoopIteration* aufgerufen.
Dies geschieht iterativ, solange bis die *GLSurfaceVIew* gestoppt wird. In der *mainLoopIteration* wird die Kamera positioniert, Eingaben abgefragt, die Bewegungs- und Positionskontrolle durchgeführt, die Lichteffekte gesetzt und das zeichnen durch OpenGL veranlasst.

4.1.3.1 LevelBuilder

Das Spiel ist gegliedert in mehrere Levels. Jedes Level muss dafür in der oben erwähnten *setup*-Methode erstellt werden. Es benötigt dazu sowohl eine zu Grunde liegende Datenstruktur eines Labyrinthes als auch mehrere Objekte die dieses Labyrinth in eine dreidimensionale Welt verwandeln. Dies realisiert die Klasse *LevelBuilder*.

Das Labyrinth Die zu Grunde liegende Datenstruktur der Welt, ist das Labyrinth. Die Bibliothek *GridMaze* stellt die nötigen Klassen und Methoden für die Labyrinthgenerierung mittels dem *depth-search* Ansatz zur Verfügung und erfüllt die in FA #1 und FA #2 definierten Anforderungen. Die Klasse *SimpleGridMaze* realisiert die Datenstruktur des Labyrinthes. Die *SimpleGridMazes* werden durch ein Tripel (Dimension X, Dimension Y, *Seed*) eindeutig bestimmt. Die Dimension X ist die Anzahl der Knoten in Richtung der X-Achse, analog dazu die Dimension Y. Der *Seed* bestimmt die zufällige Auswahl der unbesuchten Nachbarknoten in der Tiefensuche. Mit den Aufrufen *hasEast(x,y)*, *hasNorth(x,y)*, *hasWest(x,y)* und *hasSouth(x,y)* lässt sich die Existenz der entsprechenden Kante eines Knotens (x,y) überprüfen. Für den Anfangs- und Zielpunkt werden die zwei am weitesten entfernten Punkte des Labyrinthes mit dem Dijkstra-Algorithmus berechnet.

***Texture*-Klasse** Der Aufbau einer digitalen Welt benötigt Objekte aus denen die Welt besteht. In dieser Arbeit gibt es die Decke, Wände, Böden und weitere Objekte. Zur Erstellung dieser Objekte dient die Hilfsklasse *Texture*. Dazu nimmt die *Texture*-Klasse einen zweidimensionalen *float-Array* entgegen, der vier *Vertices* mit je drei Koordinaten beinhaltet und wandelt diesen in einen für OpenGL interpretierbaren *direct FloatBuffer* um. Des Weiteren stellt die *Texture*-Klasse die Methode *setTexture(Bitmap bitmap, int quantity)* zur Verfügung, die es ermöglicht eine *Bitmap* für die Texturierung der Oberfläche zu übergeben. Mit *quantity* wird die Anzahl der Texturwiederholungen für die Oberfläche angegeben. Bei $Quantity = 1$ wird mit der Option *GL_CLAMP_TO_EDGE* texturiert und die (s,t)-Koordinaten der jeweiligen Vertices auf (0,0), (0,1), (1,0) und (1,1) gesetzt. Für $Quantity > 1$ werden die (s,t)-Koordinaten der Vertices analog von (0,0) bis (quantity, quantity) gesetzt und für die Texturierung wird *GL_REPEAT* verwendet. Zuletzt bietet die *Texture*-Klasse eine *draw*-Methode, die das Zeichnen des erstellten *Meshes* inklusive der Textur veranlasst. Von der *Textur*-Klasse erben nun die *Wall*, *Floor* sowie die *WorldObject*-Klasse. Jeder dieser Klassen lädt eine gewünschte *Bitmap* aus dem *Drawables*-Ordner von *Android* und setzt entsprechend die Textur. Sämtliche Texturen wurden dabei von *CGTextures* zur Verfügung gestellt und zur weiteren Verarbeitung freigegeben [CGT13].

Levelaufbau Um die Anforderung FA #4 zu erfüllen, muss das Labyrinth in eine dreidimensionale Umgebung abgebildet werden. Den Levelaufbau steuert die Klasse *LevelBuilder*. Er verwaltet die Objekte der Welt, realisiert die Matrixtransformationen und veranlasst das Zeichnen der Objekte. Dafür müssen zuerst die Koordinaten der Objekte gesetzt werden. Zunächst Werden die Höhe des Levels (*levelHeight*) und die Knotenlänge (*knotLength*), die die Entfernung der Knoten und somit die absolute Größe des Levels bestimmt, final festgelegt. Des Weiteren muss dem *LevelBuilder* ein *SimpleGridMaze* sowie Anfangs- und Zielkoordinaten übergeben werden. Um nun die Wände anlegen zu können, müssen die Koordinaten für die *Texture*-Klasse gesetzt werden. Es werden zunächst die Koordinaten für vier Wände und einen Boden in einen für die *Texture*-Klasse interpretierbaren, zweidimensionalen FloatArray geschrieben und den entsprechenden Unterklassen von *Texture* zur Verarbeitung übergeben. Mit den angegebenen Koordinaten wird ein Quader ohne Decke in die Ecke des Koordinatensystems von OpenGL erstellt, mit der Länge und Breite *knotLength* und der Höhe *levelHeight* (siehe Abbildung 4.4).

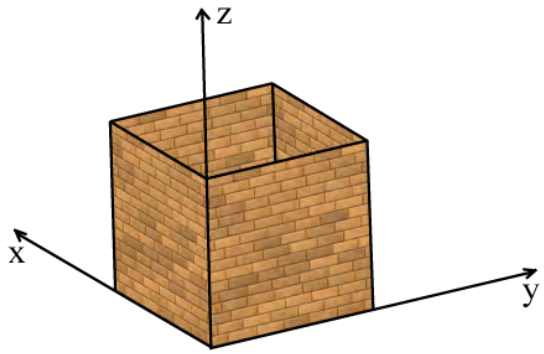

Abbildung 4.4: Quader als Basis des Levelaufbaus

Das zweidimensionale Labyrinth wird nun bei jedem Zeichenaufruf mit zwei for-Schleifen durchlaufen. Zu jedem Durchlauf der inneren Schleife wird eine Matrix geladen und eine Matrixverschiebung entsprechend der Knotenlänge vollzogen. Somit wird der Quader auf dem ganzen Graphen verschoben. Um nun nur dort Wände zu zeichnen, wo keine Verbindung bestehen, werden nach jedem Schleifendurchlauf die Verbindungen in die Süd, und West Richtung betrachtet. Gibt es keine Verbindung, so wird zusätzlich zum

Boden die entsprechende Süd- bzw. Westwand des Quaders gezeichnet (Vgl. Listing 4.3). Zusätzlich wird immer dann, wenn sich der Quader am nördlichen bzw. öslichen Rand befindet, dort die enstprechende Wand des Quaders gezeichnet. Die Decke ist ein Rechteck, mit den Ausmaßen des Labyrinthes. Sie wird mit *GL_Repeat* texturiert. Dabei sind (s,t)-Koordinaten der Eckpunkte (dimX,dimY). So erscheint über jeden Knoten die Decken Textur in der Größe *knotLength*. Somit wird ein Grundgerüst des Labyrinthes erstellt.

Listing 4.3: simpleDraw

```
1 gl.glPushMatrix();
2 gl.glTranslatef(i * knotLength, j * knotLength, 0);
3         ...
4 if (!maze.hasSouth(i, j)) {
5                                                 ...
6                 southWall.draw();
7         }
8         ...
9 gl.glPopMatrix();
```

Besonderheiten Das entstandene Grundgerüst muss nun für die Erfüllung von NFA #9 erweitert werden. Um also eine abwechslungsreiche Spielwelt zu erstellen, muss es unterschiedliche Wände, Böden und zusätzliche Objekte geben. Als zusätzliche Objekte gibt es eine Leiter, ein Spinnennetz und kleine Totenschädel. Zusätzliche Wände sind eine leicht eingebrochene Wand, eine Wand mit einem ägyptischen Bild, eine Wand mit Hieroglyphen, eine Wand mit Totenköpfen verziert sowie eine Wand mit drei großen Totenköpfen. Weitere Böden sind ein Boden mit einem leicht vergrabenen Skelett sowie ein Boden mit drei Totenköpfen. (siehe Abbildungen 4.5, 4.6, 4.7)

Diese Besonderheiten müssen nun im Level verteilt werden. Diese Verteilung soll laut FA #4 automatisch geschehen. Es muss also NFA #9 und FA #4 gemeinsam erfüllt werden. Um das zu realisieren, wird sich hier der diskreten Mathematik bedient. Es werden alle Nichtverbindungen (also dort wo Wände im Grundgerüst sind) einen Gruppe von Zahlen

Abbildung 4.5: Besondere Wände

Abbildung 4.6: Besondere Böden

Abbildung 4.7: Weitere Objekte der Welt

zugeordnet. Zusätzlich werden alle besonderen Wände einer Zahlengruppe zugeordnet. Gehört eine besondere Wand zur selben Zahlengruppe wie eine Nichtverbindung, so wird diese anstatt einer normalen Wand gezeichnet. Analog dazu die besonderen Objekte und Böden mit den Knotenpunkten. Umso willkürlicher die Zahlengruppen, umso willkürlicher erscheinen die besonderen Elemente. Um zusätzlich zu jedem anderen Level eine andere Verteilung zu erhalten, wird die Einteilung anhand des Level *seeds* bestimmt. Das ermöglicht auch, dass das gleiche Level, stets die Wände gleich verteilt. Listing 4.4 erweitert das Listing 4.3, und zeigt die Einteilung einiger besonderen Wände in die Zahlengruppen.

Listing 4.4: Einteilung in Zahlengruppen

```
1  if (!maze.hasSouth(i, j)) {
2          if ((i * j) % 5 == maze.getSeed() % 4
3                          && (j + i) % 3 == maze.getSeed() % 2) {
4          fontWall.draw();
5          } else {
6                  if ((i * j) % 7 == maze.getSeed() % 4
7                  && (j + i) % 3 == maze.getSeed() % 3) {
8                                  skeletWall.draw();
9                  } else {
10                         if ((i * j) % 7 == maze.getSeed() % 5
11                         && (j + i) % 3 == maze.getSeed() % 2) {
12                                         brokenWall.draw();
13                         } else {
14                                         southWall.draw();
15 }}}}
```

4.1.3.2 Bewegungssteuerung

Wie in den Anforderungen erwähnt, soll die Bewegung durch Swipegesten gesteuert werden. Dabei soll ein Swipe nach vorne oder hinten einen Schritt vorwärts bzw. rückwärts

bewirken und ein seitlicher Swipe eine Drehung von 90 Grad. Da OpenGL Thread- und nicht Event-basiert ist, wird in jedem Durchlauf des *GameLoops* die Berührungsspanne betrachtet. Übersteigt die Bewegungsspanne einen gewissen Wert, wird eine Bewegung veranlasst.

4.1.3.3 Positionskontrolle

Die Positionskontrolle wird in dieser Arbeit durch die Klasse *Position* realisiert. Sobald eine Bewegung veranlasst wird, wird Richtung und Art der Bewegung, der *Positions*-Kontrolle mitgeteilt. Es gibt fünf Arten von Bewegungen:

- Schritt vorwärts
- Schritt rückwärts
- Drehung links
- Drehung rechts
- Negationsbewegung

Die Negationsbewegung simuliert ein Kopfschütteln, wenn man sich auf eine Wand zu bewegt. Auch hier ist wieder zu beachten dass OpenGL nicht Event-basiert arbeitet. Das heißt es wird keine Bewegung initiiert und auf deren Ausführung gewartet. Um eine Event-basierte Eingabe zu simulieren, besitzt die *Position*-Klasse Zustände die gesetzt werden können. Sollte sich die *Position* in einem Bewegungszustand befinden, so lässt sie keine Veränderungen zu bis die aktuelle Bewegung zu Ende ist und sie sich wieder im Initialzustand befindet. Abbildung 4.8 zeigt die Möglichen Zustände der *Position*-Klasse. Unterer Aufruf, zeigt die Negationsbewegung, die sich in fünf einzelne Zustände untergliedern lässt.

Um die Bewegungen voranzutreiben, wird nach jedem *GameLoop* die *Position*-Klasse mit *position.refresh(DeltaTime)* aktualisiert, mit der *DeltaTime* als Übergabeparameter. Die Positionskontrolle verwaltet nun zwei Positionsinformationen. Es wurde erläutert, dass das Labyrinth als Graph zugrunde liegt, der als zweidimensionalen Array gespeichert

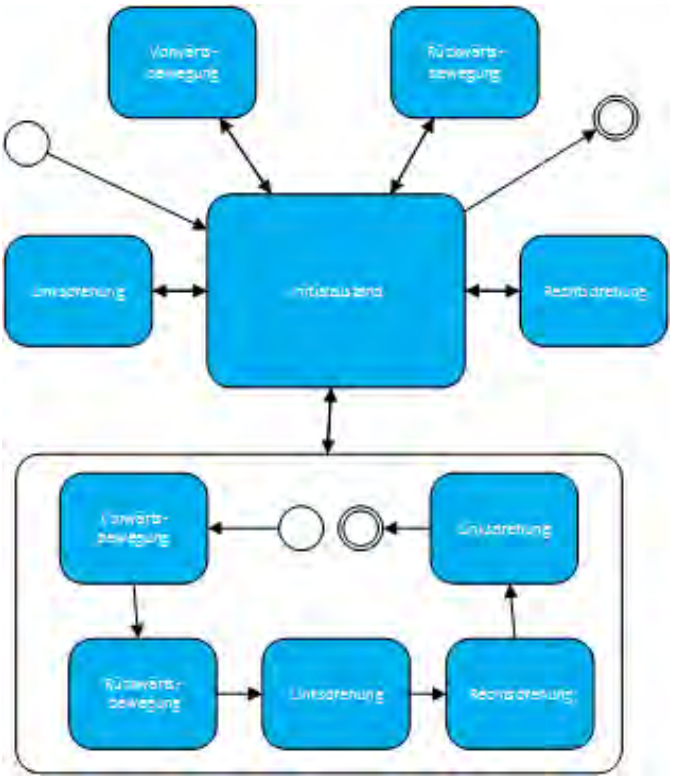

Abbildung 4.8: Zustände der *Position*-Klasse

wird. Die erste Positionsinformation bezieht sich auf den Knoten, in dem der Spieler sich im Labyrinth befindet. Sie wird als ganze Zahl gespeichert. Zum Anderen hat der Spieler eine exakte Position im Raum. Bei einem Schritt in eine bestimmte Richtung muss zuerst die Blickrichtung betrachtet werden. Ist der Blick nach Norden/Süden gerichtet, wird bei einem Schritt vorwärts die Y-Koordinate um Eins erhöht/verringert. Bei einem Schritt rückwärts genau umgekehrt. Bei einem westlich/östlich gerichteten Blick wird analog zur X-Koordinate vorgegangen. Allerdings geschieht dass nur, wenn in dieser Richtung eine Verbindung zwischen den Knoten existiert. Ansonsten wird die Negationsbewegung ausgelöst. Um die Illusion einer fließenden Bewegung zu simulieren, muss zusätzlich nach jedem Zeichenaufruf eine exakte Position zwischen den Knoten berechnet werden. Dafür wird bei einer Positionsänderung zuerst Endpunkt der Bewegung festgelegt. Anschließend muss die Schrittlänge (entspricht der Knotenlänge) durch die Anzahl der Bilder geteilt werden, die während dem Weg von Start- und Endpunkt gezeichnet werden. Da es nicht bekannt ist, wie viele Bilder in dieser Zeit gezeichnet werden, wird sich der *DeltaTime* bedient, die Zeit, die seit dem letzten Zeichenaufruf vergangen ist. Die *DeltaTime* wird mit der Schrittlänge multipliziert und anschließend mit dem gewünschten Zeitintervall in dem der Schritt ausgeführt werden soll. Dies wird dann solange zur Position der entsprechenden Dimension addiert bzw. subtrahiert, bis die Endposition erreicht wurde (siehe Listing 4.5).

Listing 4.5: Berechnung der Vorwärtsbewegung

```
1 posX += deltaTime * knotLength * movingSpeed;
2 if (posX >= finPosX) {
3         posX = finPosX;
4         movingForward = false;
5 }
```

Ähnlich wird die Drehung realisiert. Dazu wird zuerst die finale Blickrichtung gesetzt. Anschließend werden die 90 Grad der Drehung für die einzelnen Teilbilder zerlegt. Hierfür werden die 90 Grad mit der *DeltaTime* multipliziert und das mit der gewünschten Drehgeschwindigkeit multipliziert. Der Teilwinkel wird anschließend zu dem aktuellen Winkel addiert bzw. subtrahiert, bis der finale Blickwinkel erreicht wurde. Da die Kamera

in der *MainLoopIteration* einen Blickpunkt anstatt eines Blickwinkels benötigt, muss dieser noch berechnet werden. Dafür geht man davon aus, dass dieser Blickpunkt sich auf einer Kreisbahn um die aktuelle Position dreht. Somit lassen sich die Koordinaten des Blickpunktes mit $x_{look} = Radius * cos(Blickwinkel) + x_{pos}$ und $y_{look} = Radius * sin(Blickwinkel) + y_{pos}$ berechnen. Nun kann die *MainLoopIteration* die benötigen Koordinaten von der *Position*-Klasse abfragen, um die Kamera zu setzen.

4.1.3.4 Perspektive

Um eine authentische Atmosphäre zu schaffen, bedarf es auch einer glaubwürdigen Perspektive. Dafür werden die Einstellungen der Perspektive in der *Projektionsmatrix* gesetzt, als auch der POV und Blickwinkel in der *ModelView*-Matrix festgelegt. Die Einstellungen der Perspektive werden wie in Kapitel 2 erörtert mit *gluPerspetive* gesetzt. In dieser Arbeit wird der Aufruf wie folgt getätigt:

Listing 4.6: Festlegung der Perspektive

```
1 GLU.gluPerspective(gl, 105, aspectRatio, 0.2f, stepLength*20);
```

Der *aspectRatio* ist das Verhältnis der Fensterbreite, zur Fensterhöhe, um die Darstellung nicht zu verzerren. Die *Near Clipping Pane* beträgt 0.2f. Ursprünglich war diese höher, durch die Implementierung der Bewegung auf eine Wand hin, wurde sie jedoch herabgesetzt, um nicht durch die Wand blicken zu können. Die *Far Clipping Pane* wurde empirisch durch zahlreiche Testspiele bis auf das 20 fache der Schrittlänge festgelegt. Der Winkel des Sichtfeldes Entstand ebenfalls durch Empirie.

Man sieht in den Abbildungen 4.9 bis 4.13 die Einstellungen der Perspektive mit den jeweiligen Grad für das Sichtfeld. Es wurde sich in dieser Arbeit für 105 Grad entschieden, da hier noch keine Verzerrung des Umfeldes erkennbar ist, jedoch die komplette Textur einer Wand sichtbar ist, wenn man sich direkt vor dieser befindet. Ein weiterer Punkt der für die Perspektive zu beachten ist, ist die Festlegung der Kamera. Wie in Kapitel 2 beschrieben, dient dafür der Methodenaufruf *glLookAt*. *glLookAt* wird in jeder *GameLoop*-Iteration ausgeführt. Für die Übergabeparameter werden die (x,y)-Positionskoordinaten sowie die (x,y)-Koordinaten des Betrachtungspunkt aus der *Position*-Klasse geladen.

Abbildung 4.9: Sichtfeld mit 40 Grad

Abbildung 4.10: Sichtfeld mit 67 Grad

Abbildung 4.11: Sichtfeld mit 90 Grad

Abbildung 4.12: Sichtfeld mit 105 Grad

Die Höhe der Kamera sowie die Höhe des Betrachtungspunktes werden in Abhängigkeit der *levelHeight* gesetzt. In dieser Arbeit wird der Wert für die Kamerahöhe auf *levelHeight**0.8 gesetzt um die menschliche Augenhöhe zu repräsentieren und der Wert *levelHeight*/2 für die Höhe des Blickpunktes, um eine schräge Sicht von oben nach unten zu erzeugen. Der *up*-Einheitsvektor ist in dieser Arbeit (0,0,1). Dadurch zeigt die y-Achse in die Tiefe, die x-Achse in die Breite und die z-Achse in die Höhe. Da die Koordinaten des *SimpleGridMaze* in einem (x,y)-Koordinatensystem gegeben sind, soll dadurch Verwirrung vermieden werden.

4.1.3.5 Lichtgestaltung

Es soll nach FA #5 eine Lichtgestaltung die Welt authentischer machen. In dieser Arbeit existieren zwei Lichtquellen. Ein *ambientes* Licht und ein *Spotlight*. Für beide Lichtquellen wird je ein eindimensionales Floatarray mit vier Einträgen für den *RGBA*-Farbraum benötigt. Mit dem *ambienten* Licht wird hier die Helligkeit gesteuert. Je nach gewünschter Helligkeit wird der Wert des *ambienten* Lichtes von {0.1f, 0.1f, 0.1f, 1f} bis {1f, 1f, 1f, 1f} gestezt. Man beachte, dass hier die Werte für Rot, Grün und Blau identisch bleiben, um die Farben der Texturen nicht zu verzerren. Um den *Spotlight* zu setzen wählt man eine der acht Lichtquellen von OpenGL aus. Danach setzt man die Farbwerte für die einzelnen Lichtarten. Da in dieser Arbeit Materialeigenschaften nicht berücksichtigt werden, genügt es allen drei Lichtarten denselben Farbraum zu übergeben. Zusätzlich wird die Postion des Lichtes sowie ein Einheitsvektor für die Lichtrichtung benötigt. Als Position erhält das Licht zunächst den Ursprung. Der Richtungsvektor erhält den Wert (0.6f, 0, -0.8f). Dieser zeigt nun schräg nach unten. Mit Matrixverschiebung und Matrixrotation wird das Licht vor jeder *GameLoop*-Iteration entsprechend auf die Postion der Kamera gesetzt, mit Ausrichtung in das Blickfeld des Spielers. Abschließend muss der Ausfallwinkel des Lichtkegels, der sogenannte *CUT OFF* bestimmt werden.

Listing 4.7: Aufrufe zur Beschreibung des *Spotlights*

```
1 gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_POSITION, lightPosition, 0);
2 gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_SPOT_DIRECTION,
3                                         lightDirection, 0);
```

```
4 gl.glLightf(GL10.GL_LIGHT0, GL10.GL_SPOT_CUTOFF, 45.0f);
5 gl.glEnable(GL10.GL_LIGHT0);
```

4.1.4 Items

Die Anforderungen des Spieles beinhaltet mehrere Items (FA #6). Es wird sowohl ein *Kompass*, ein *Zielpfeil* als auch eine *Karte* gefordert (FA #10, FA #11, FA #12). Um die Items freizuschalten, soll es eine spielinterne Währung geben, sogenannte *Credits* (FA #9).

4.1.4.1 Credits

Die *Credits* dienen zur Freischaltung der Items. Um *Credits* zu erhalten, müssen Goldbarren, die zufällig im Level verteilt sind, aufgesammelt werden. Ein Goldbarren fügt sich aus fünf Texturflächen zusammen und hat die Form eines Trapezprismas mit einem Neigungswinkel von 80 Grad. Dabei nimmt jede Fläche ein goldene, schwarz, umrahmte Textur an. Die *GoldBar*-Klasse besitzt je Fläche eine Instanz der *Texture*-Klasse, sowie Positionsinformation und einen Wert. Außerdem besitzt die Klasse eine Abfrage ob der Spieler sich auf dem Feld des Goldbarrens befindet. Um mehrere Goldbarren im Level zu verteilen, werden im *LevelBuilder* solange neue Goldbarren mit einer Position versehen und in einer Liste gespeichert, bis die gewünschte Anzahl an Goldbarren erreicht ist. Nach jedem *GameLoop* wird die Position des Spielers, mit der der Goldbarren verglichen. Sollte die Position des Spielers identisch mit der eines Goldbarrens sein, so wird der Goldbarren aus der Liste entfernt und der zugehörige Wert dem *Creditkonto* gut geschrieben. Das *Creditkonto* ist mit den *SharedPrefernces* von *Android* realisiert.

4.1.4.2 Kompass

Der *Kompass* ist ein Item das die Himmelsrichtungen Nord, West, Süd und Ost kennzeichnet und sich der Spielerrotation entsprechend, dreht. Somit zeigt der *Kompass* stets

Abbildung 4.13: Sichtfeld mit 140 Grad

Abbildung 4.14: Darstellung der Items, vor und nach der Aktivierung

in die richtige Richtung. Der *Kompass* bedient sich der *Texture* Klasse für seine Oberfläche. Um stets im Blickfeld des Spielers zu sein, benötigt es wieder entsprechende Matrixtransformationen. Dafür soll der *Kompass* sich auf einer Kreisbahn um den Spieler befinden. Wie im Abschnitt zu Positionskontrolle gezeigt, dient dazu die Formel $x_{kompass} = Radius * cos(Blickwinkel) + x_{pos}$ und $y_{kompass} = Radius * sin(Blickwinkel) + y_{pos}$. Somit "kreist" der *Kompass* zentriert vor der Kamera. Damit der *Kompass* seitlich von dem Spieler versetzt erscheint, muss der Mittelpunkt der Kreisbahn um der sich der *Kompass* dreht, seitlich zum Spieler versetzt werden. Deshalb wird die Positionskoordinate wie folgt verändert:
$x_{Kompass} = Radius * cos(Blickwinkel) + (x_{pos} + sin(Blickwinkel) * Seitenverschiebung)$
und
$y_{Kompass} = Radius * sin(Blickwinkel) + (y_{pos} - cos(Blickwinkel) * Seitenverschiebung)$.
Der *Kompass* selbst rotiert dabei nicht, sondern bleibt stets gleich ausgerichtet. Durch die Drehung der Kamera erscheint es, als würde der *Kompass* sich drehen. Für eine bessere Ausrichtung im Raum, wird der *Kompass* zusätzlich noch um 35 Grad in Richtung der Y-Achse des Bildschirmes und um 15 Grad in Richtung der X-Achse des Bildschirms gedreht. Dazu wird der Einheitsvektor der entsprechenden Rotationsachse benötigt. Für die Rotation in Richtung der X-Achse des Bildschirms muss der Einheitsvektor in die Blickrichtung des Spielers zeigen, für die Rotation in Richtung der Y-Achse des Bildschirms 90 Grad zur Blickrichtung des Spielers. Also sind die Koordinaten des Einheitsvektors $(x, y, z) := (sin(Blickwinkel), cos(Blickwinkel), 0)$ bzw. $(x, y, z) := (sin(Blickwinkel + 90), cos(Blickwinkel + 90), 0)$.

4.1.4.3 Zielpfeil

Der *Zielpfeil* ist ein Item, das stets in die Richtung des Ausganges zeigt, indem er sich entsprechend dreht. Analog zum *Kompass*, "kreist" der *Zielpfeil*, seitlich versetzt um den Spieler. Damit der *Zielpfeil* stets in Richtung des Zieles zeigt, muss sich der *Zielpfeil*, entsprechend dem Verhältnis der Spielerposition zur Zielposition, rotieren. Für die Winkelberechnung der Rotation, denke man sich ein rechtwinkliges Dreieck, wie in Abbildung 4.15.

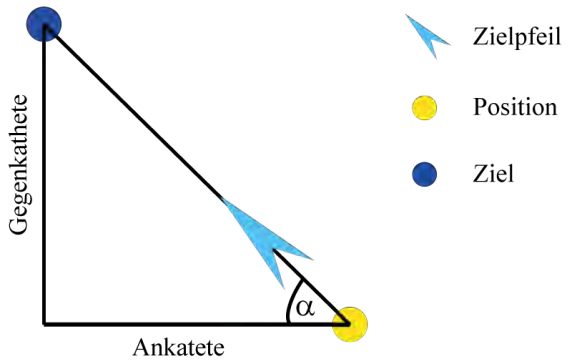

Abbildung 4.15: Rechtwinkliges Dreieck zur Winkelberechnung des *Zielpfeiles*

Die Ankathete wird mit $ak = |x_{pos} - x_{ziel}|$ und die Gegenkathete mit $gk = |y_{pos} - y_{ziel}|$ berechnet. Der Rotationswinkel ergibt sich aus $\alpha = tan^{-1}(gk/ak)$.

4.1.4.4 Karte

Die *Karte* ist eine kartographische Aufsicht auf das Labyrinth. Sie dient der Orientierungshilfe. In ihr sind die Wände des Labyrinthes als schwarze Balken dargestellt. Start- bzw. Endpunkt sind mit farbigen Kreise markiert und die aktuelle Position wird durch ein farbiges Dreieck vermerkt. Für die *Karte* werden zwei Klassen benötigt. Das *MapRL* welche von *RelativeLayout* erbt, sowie die *MapView* die von *View* erbt. Das *MapRL* dient der Gestaltung der *Karte* und kann in eine *Activity* eingebettet werden, während die *MapView* das Zeichnen der *Karte* realisiert. Das *MapRL* besitzt ein aufgeschlagene Papyrusrolle als Hintergrund, zwei *TextViews* zur Kennzeichnung des Start- und Endpunktes und bindet die *MapView* ein. Um die *Karte* in der *MapView* zu Zeichnen, wird die Methode *onDraw* verwendet, die ein *Canvas* als Zeichenfläche liefert. Zusätzlich werden das *SimpleGridMaze* und die Postion benötigt. Für die schwarzen Balken wird die Methode *drawRect* verwendet. *drawRect* benötigt je eine X-Koordinaten für die linke und rechte Seite eines Rechteckes sowie je eine Y-Koordinate für die obere und untere Seite. Des Weiteren wird eine Instanz der Klasse *Paint* benötigt. *Paint* dient zur Beschreibung der Eigenschaften des zu zeichnenden Objektes, zum Beispiel für Farbe, Linienstärke und Fülloptionen. Für die Koordinaten der Balken, werden zunächst Länge

und Breite berechnet. Die Länge der Balken ergibt sich wie folgt:
$Balkenlänge := min(Breite_{mapView}/LabyrinthDimension_x,$
$Höhe_{mapView}/LabyrinthDimension_y)$.
Die Balkenbreite ist ein Drittel der Balkenlänge. Analog zum Levelaufbau wird nun das zweidimensionale Array des Labyrinthes durchlaufen und überall dort, wo keine Kante nach Norden bzw. Osten verläuft, ein schwarzer Balken in entsprechender Ausrichtung gezeichnet. Anschließend werden noch die Wände für den südlichen bzw. westlichen Rand des Labyrinthes gezeichnet. Zu beachten ist hier, dass die Y-Achse im Koordinatensystem des *Canvas*, in negativer Richtung zur Y-Achse des Koordinatensystems von *OpenGL* verläuft. Daher muss das Array des Labyrinthes für die Y-Dimension in umgedrehter Reihenfolge durchlaufen werden. Da Canvas kein Zeichnen von Polygonen unterstützt, wird das Dreieck der Spielerposition mittels einen Pfad realisiert. Hierauf ist zu achten, dass Startpunkt und Endpunkt identisch sind.

Listing 4.8: Pfadgenerierung für das Positionsdreieck

```
1 posPath.moveTo(-lineWidth/4, -lineWidth/4);
2 posPath.lineTo(-lineWidth/4, lineWidth/4);
3 posPath.lineTo(lineWidth/3,0);
4 posPath.lineTo(-lineWidth/4, -lineWidth/4);
5 posPath.close();
```

Anschließend erlaubt *Canvas*, ähnlich zu *OpenGL*, eine Verschiebung und Rotation des Dreieckes. Zuletzt werden noch Start- und Zielpunkt mittels *drawCircle(float cx, float cy, float radius, Paint paint*) gezeichnet.

4.1.4.5 Einbindung in die Activity

Das Spiel wird, für die Verwendung in dem Fenstersystem von Android, in eine *Activity* eingebunden. Da die *GLSurfaceView*, die die Zeichenoberfläche für OpenGl darstellt, in dem *Fragment GameFragment* implementiert ist, muss nun diese in eine Activity platziert werden. Das *GameFragment* stellt sowohl das reine Spiel, als auch die beiden Items *Kompass* und *Zielpfeil* dar. Es soll jedoch noch eine Leiste geben die Informa-

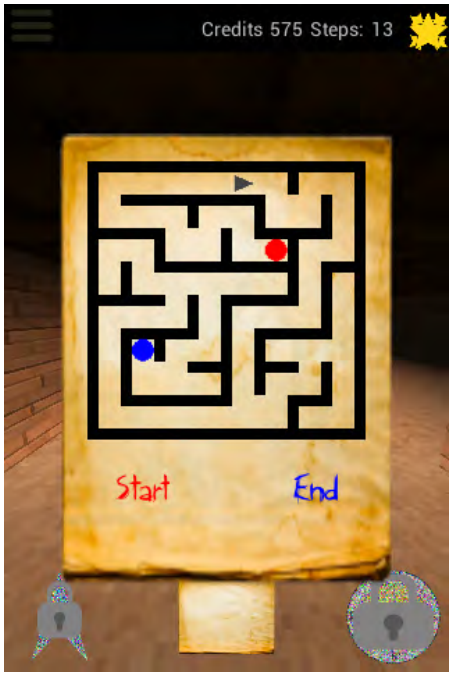

Abbildung 4.16: Ansicht der *Karte*

tionen zu dem Spiel bietet, wie durchgeführte Schritte, vorhandene *Credits* als auch ein aktuelles *Rating* um FA #8 zu erfüllen. Diese Informationen werden Oberhalb des Spieles dargestellt und durch ein *RelativeLayout* realisiert. Die Informationsleiste wird am oberen Rand in die *GameActivity* eingefügt und darunter das *GameFragment*. Um die Items aktivieren zu können, wird ein weiteres *RelativeLayout* am unteren Ende der *GameActivity* eingebettet und mit Buttons für die einzelnen Items versehen. Diese *RelativeLayout* wird über das *GameFragment* platziert, sodass es dieses überdeckt (siehe Abbildung 4.17).

Abbildung 4.17: Aufbau der GameActivity

Damit das *GameFragment* mit der *GameActivity* kommunizieren kann, wird ein *GameChangeListener* verwendet, über den in der *mainLoopIteration* Änderungen des Spieles an die *GameActivity* übergeben werden kann.

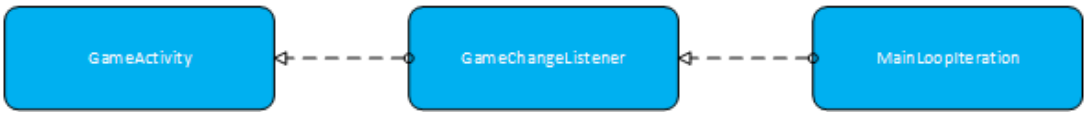

Abbildung 4.18: Kommunikation des der GameLoop mit der Activity

4.1.5 Alternativer Levelaufbau und Vergleich

Eine alternative Möglichkeit das Grundgerüst der Levels aufzubauen, ist es, nicht lediglich vier Wände zu erstellen und durch Matrixverschiebungen an die gewünschte Position zu zeichnen, sondern eine eigene *Wall*-Instanz für jede Nichtverbindung zu erstellen. Dies wird analog zum vorherigen Aufbau realisiert, mit dem unterschied dass bereits im *setup* alle Wände mit dem entsprechenden absoluten Koordinaten erstellt werden. In der *MainLoopIteraionen* werden diese dann ohne Matrixverschiebungen gezeichnet. Analoges gilt für die Besonderen Objekte. Dieser alternative Levelaufbau war der ursprüngliche Ansatz, wurde jedoch für eine bessere Performance überarbeitet (NFA #10).

Vergleich

Die zwei Herangehensweisen des Levelaufbaus sollen nun Hinsichtlich der Performance verglichen werden. Aspekte die dazu betrachtet werden, ist sowohl die Dauer des Levelaufbaus als auch die CPU und GPU Auslastung währende des *GameLoops*. Um die Perfomance zu untersuchen wurden verschiedene große, quadratische Level mit beiden Ansätzen erstellt. Zum untersuchen des Zeitverhaltens, wurde ein Level je Größe zehn mal gestartet, die durchschnittliche Dauer des Levelaufbaus gemessen und tabellarisch festgehalten (siehe Tabelle 4.1). Die Tabelle zeigt einen signifikanten Zeitunterschied zwischen den verschiedenen Levelgenerierungen. Während die Dauer die Level via Matrizentransformation zu realisieren linear zur Levelgröße ansteigt, wächst die Zeit das Level absolut zu gestalten exponentiell für größere Level. Das ist damit zu erklären, dass die Matrizenverschiebung es erlaubt die Anzahl an Objektinstanzen erheblich zu reduzieren und folgernd den benötigten Speicherbedarf im RAM verringert.

Eine Weitere Betrachtung der Perfomance ist die Auslastung der CPU und GPU während dem *GameLoop*. Hierzu diente die Android Developer Option zur Anzeige der CPU und GPU Auslastung. Abbildung 4.19 zeigt zwei Screenshots mit den aktivierten Optionen. Links wird das Spiel statisch aufgebaut und rechts durch Hilfe der Matrizentransformationen realisiert. Die Auslastung der CPU ist im rechten, oberen Ecke der Screenshots zu sehen. Dabei steht an zweiter stelle der Prozess des Spieles. Die Linien am unteren Bildschirmrand visualisieren die Auslastung der GPU. Dabei wird die Verarbeitungsdauer der letzten 128 Bilder gezeigt. Die blaue Linie zeigt die Zeit, die für das reine Zeichnen verging. Die rote Linie betrifft die Dauer die für die Erstellung der Befehle benötigt wurde und die orangene Linie die Zeit die Ausführung der Befehle. Die grüne Linie skaliert 16ms, die für eine Bildwiederholungsfrequenz von 60 fps der meisten heutigen Bildschirme, nicht überboten werden sollte. Erfahrungen haben gezeigt, dass das reine zeichnen bei beiden Methoden der Levelgenerierung in etwa identisch sind, da die gezeichneten Welten sich nicht unterscheiden. Die Erstellung der Befehle benötigte für die Matrizentransformation etwas länger, dass dank Hardwarebeschleunigung jedoch vernachlässigbar ist. Einen deutlichen Unterschied erkennt man bei der Ausführungszeit der Befehle sowie der CPU-Auslastung. Das ist erneut auf die unterschiedliche Menge zu verarbeitenden Positionsdaten der zwei Verfahren zurückzuführen. Erfahrungen haben diesen Eindruck beim spielen sehr großer Level bestätigt. Während bei dem Levelaufbau durch Matrizenverschiebung noch ein flüssiges Bild beobachtet wurde, war bei dem alternativen Verfahren bereits ein Ruckeln des Bildes erkennbar.

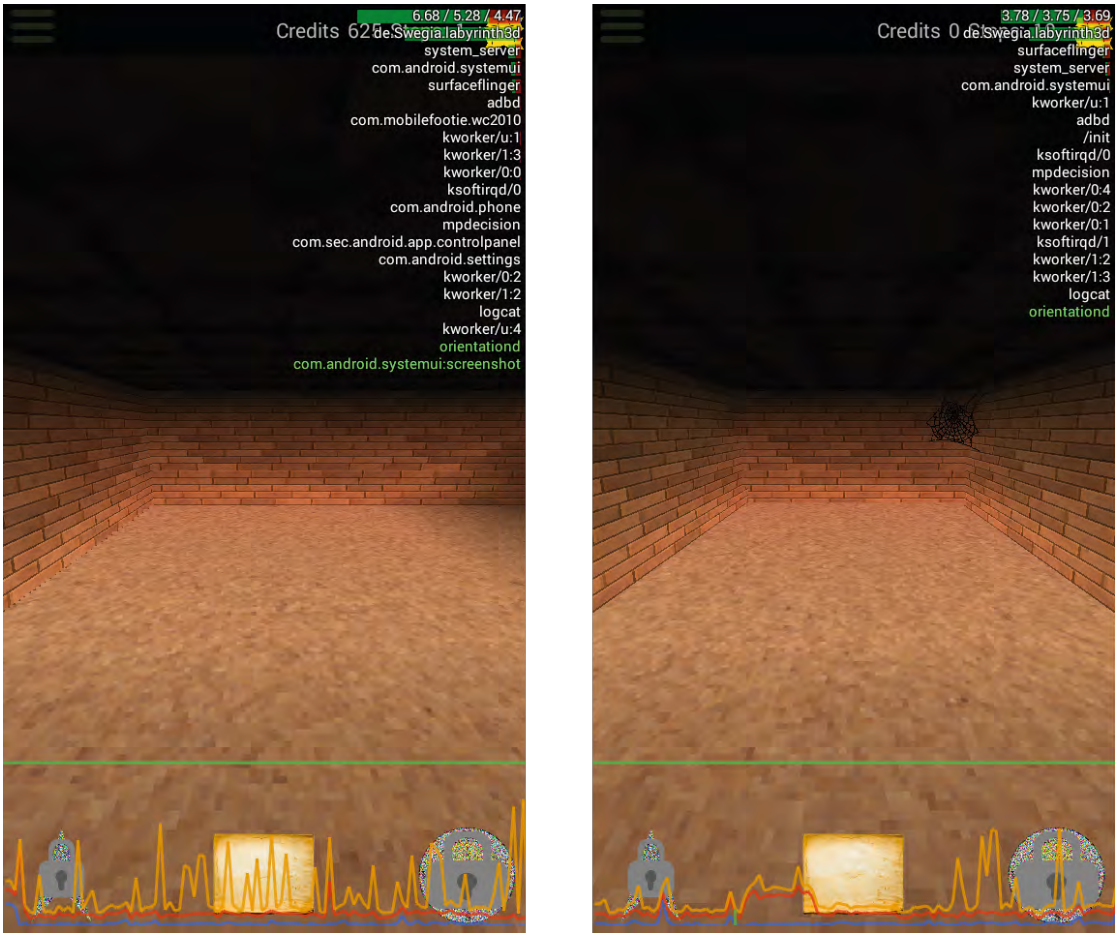

Abbildung 4.19: Peformance Vergleich der verschiedenen Levelgestaltungen

Tabelle 4.1: Tabellarische Erfassung des Durchschnittlichen Zeitverhaltens in Sekunden

Ansatz / Levelgröße	5×5	10×10	15×15	20×20	25×25
Levelaufbau durch Matrixverschiebung	0,285	0,453	0,751	0,899	1,172
Absoluter Levelaufbau	0,467	1,493	3,23	6,67	9,130

4.2 Labyrinth im Kontext *Track Your Tinnitus*

Das Labyrinth soll nun eine Erweiterung der Applikation *Track Your Tinnitus* sein. Ziel des Spieles ist es dabei, möglichst die volle Aufmerksamkeit des Spielers zu erlangen und ihn somit von seinem Tinnitus ablenken zu können. In den folgenden Abschnitten wird dargestellt, wie die in Kapitel 3 definierten Anforderungen umgesetzt wurden.

4.2.1 Dialogsystem

Zunächst wird ein Dialogsystem zur Steuerung der Funktionen benötigt (siehe Abbildung 4.20). Ausgangspunkt ist ein Hauptmenü von dem die gegebenen Funktionen aufgerufen werden. Das Starten des Spiels geschieht durch einen großen zentralen Button da dieser Aufruf als besonders wichtig angesehen wird. Die weiteren Funktionen sind lediglich Erweiterungen. Deshalb wird das verwalten der Freundesliste, die Einstellungen, das Abrufen der Online Bestenliste und das Einsehen der Informationen zum Spielfortschritt durch kleinere Buttons am unteren Rand des Bildschirms ermöglicht (siehe Abbildung 4.21).

4.2.2 Anpassungen des Labyrinth für *Track Your Tinnitus*

Um den Anforderung für das Spiel im Kontext *Track Your Tinnitus* gerecht zu werden, muss das Spiel dafür entsprechend angepasst werden. Zunächst soll FA #13 betrachtet werden. Um möglichst die volle Aufmerksamkeit des Spielers zu erhalten, muss wie in FA #14 definiert, der Schwierigkeitsgrad und damit die Komplexität des Levels, sich dem Spieler anpassen. Der zugrunde liegende Algorithmus für die Erstellung des Labyrinthes basiert, wie in Kapitel 2 erörtert, auf dem Prinzip der Tiefensuche. Bestimmend für die Anordnungen der Verbindungen eines fertigen Labyrinthes, ist die Wahl der unbesuchten Nachbarn. Diese Wahl wird lediglich durch die *Seed*-Zahl beeinflusst. Dieser *Seed* lässt jedoch keinen Rückschluss auf die Komplexität des resultierenden Labyrinthes zu. Daher ist der verwendete Ansatz für die Justierung des Schwierigkeitsgrades derjenige, dass je größer das Level ist, desto schwerer ist es im Allgemeinen zu lösen. Deshalb wird

Abbildung 4.20: Dialogsystem im Überblick

in dieser Arbeit die Größe des Levels variiert, um den Schwierigkeitsgrad anzupassen. Des Weiteren muss um FA #13 zu erfüllen, der Score, wie in FA #15 erwähnt, in “gut“, “mittel“ und “schlecht“ kategorisiert werden. Die Berechnung des Scores erfolgt durch zwei Veränderliche. Die benötigten Schritte sowie die vergangene Zeit. Um den Score zu kategorisieren wird der aktuelle Score berechnet und im Verhältnis zu einem “bestcase“ Score gesetzt. Der “bestcase“ Score wird durch die Schrittanzahl des Lösungsweges sowie durch einer Dauer von 15 Sekunden (NFA #11) bestimmt. Ist der erhaltene Score höher als der “bestcase“ Score oder unterschreitet diesen nur knapp, so wird der Score als “gut“ kategorisiert. Hat der erhaltene Score nur eine gewisse Abweichung zum “bestcase“ Score, wird der Score als “mittel“ eingestuft, übersteigt er diese Abweichung, als “schlecht“. Um nun die Levelgröße entsprechend anzupassen, besitzt der Spieler einen Levelrang, der zu Beginn fünf beträgt. Bei einem “gutem“ Abschluss wird dieser erhöht, bei einem “schlechtem Abschluss“ verringert, und bei einem “mittlerem“ Abschluss bleibt dieser unverändert. Die Größe des Levels ergibt sich nun aus

Abbildung 4.21: Activity für das Haptmenü

$Dim_x = Dim_Y = (Rang + 4)/2$ um die Größe konstant, aber nicht zu stark, zu erhöhen oder zu verringern. Durch diese Anpassung der Level Größe soll nun die Spieldauer die in FA #13 geforderte Zeit approximieren.

4.2.3 Zusätzliche Features

Um dem Spiel einen zusätzlichen Reiz geben werden weitere Features angeboten. Diese werden durch die kleinen Buttons im Hauptmenü aufgerufen. Der Spielfortschritt zeigt dem Spieler eine das Spiel betreffende Statistik. Sie zeigt dem Spieler die Anzahl seiner jemals getätigten Schritten, die gespielte Zeit, aktuelle Anzahl an *Credits* sowie bereits ausgegebene *Credits*. Diese Statistik soll den Reiz das Spiel erhöhen und den Benutzer eine stärkere Verbindung zu dem Spiel geben.

Abbildung 4.22: Activity für den Fortschritt

Des Weiteren gibt es die Funktion die das verwalten und auflisten von Freunden erlaubt wie in FA #17 gefordert (siehe Abbildung 4.23). Sofern eine Internetverbindung vorhanden ist, werden die Namen aller aktuellen Freunde in einer Liste angezeigt. Zum Hinzufügen eines Freundes, wird ein Dialogfenster mit einem Klick auf das Plussymbol geöffnet. Auch hierfür benötigt es eine Verbindung mit dem Internet. Das Dialogfenster bietet ein Eingabefeld für die Suche. Eine Server suche wird initialisiert und alle, mit dem Suchbegriff verwandten Namen werden in einer Liste als *json*-Objekt zurückgegeben. Mit einem Klick auf einen Namen kann dieser zur Freundesliste hinzugefügt werden. Dabei wird nicht für jeden Benutzer eine eigene Freundesliste verwaltet, sondern eine gemeinsame für alle (siehe Abschnitt 4.2.4). Somit wird, wenn Benutzer A, Benutzer B als Freund hinzufügt, ebenfalls in der Freundesliste von Benutzer B, Benutzer A erscheinen. Damit soll der Aufwand zur Verwaltung der Freunde verringert werden, denn es muss keine Freundschaftsanfrage gestellt und akzeptiert werden und es ist ausreichend, dass einer der zwei Freunde die Freundschaft erstellt.

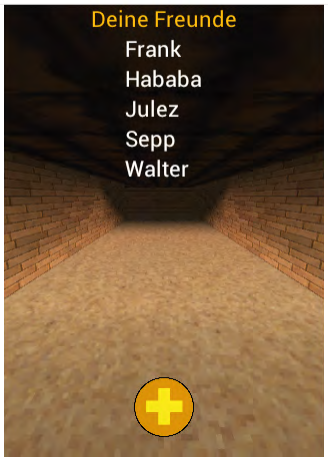

Abbildung 4.23: Activity für die Freundesliste

Wie in FA #16 definiert gibt es zu jedem Level je eine gesamte Bestenliste sowie eine Bestenliste der Freunde (siehe Abbildung 4.24). Diese lassen sich über jeweils zwei Pfeile navigieren. Zu jeder Bestenliste wird hierbei eine eigene Serveranfrage gestellt. Durch die Bestenliste soll der Spieler seine eigene Leistung besser einschätzen können. Außerdem soll das Messen mit anderen Spielern und Freunden den Reiz des Spieles erhöhen. Da sich die Komplexität der Level lediglich über die Veränderung der Levelgröße verändern lässt und es nicht bekannt ist wie komplex ein Level ist, wird als Vergleich nicht der Levelrang, sondern die Länge des kürzesten Zielweges verwendet.

Zuletzt lässt sich ein Menü für die Einstellungen öffnen (siehe Abbildung 4.25). Es lässt sich hier die Steuerung spezifizieren sowie die Sensibilität der Steuerung einstellen. Diese Einstellungen werden mit den *SharedPreferences* von Android verwaltet. Erfahrungen haben gezeigt dass die Intuitivität der Steuerung stark vom Benutzer abhängt, da vor allem Apple-User die Swipegesten in entgegengesetzter Richtung zur gewünschten Bewegung tätigen. Um eine intuitive Steuerung zu gewährleisten, bedarf es einer Einstellungsmöglichkeit der Swiperichtung. Die Sensibilität der Steuerung ist definiert durch die Entfernung die die Swipegeste zurücklegen muss, um eine Steuerung auszuführen. Bei einer hohen Sensibilität kann es sein, dass eine gewünschte Drehung, einen Schritt vor oder zurück auslöst und umgekehrt. Bei einer niedrigen Sensibilität kann die Ent-

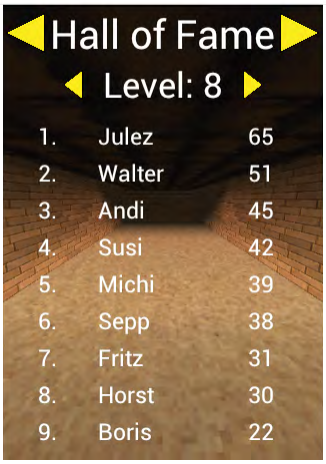

Abbildung 4.24: Activity für die Bestenlisten

fernung, die die Swipegeste zurücklegen muss um eine Bewegung zu veranlassen, als unangenehm lang Empfunden werden. Da dies stark vom verwendeten Gerät abhängt, soll die Sensibilität einstellbar sein.

4.2.4 Server

Damit eine Online-Bestenliste sowie eine Freundesliste zu realisieren ist, bedarf es einen Server der die Scores der Benutzer speichert, die Freundesliste verwaltet und mit dem das mobile Gerät kommunizieren kann (siehe Abbildung 4.26).

Der Server speichert die Daten in einer *MySQL* Datenbank. Die benötigten Informationen werden Dabei in drei Datenbank Tabellen verwaltet. Eine für die User, mit lediglich dem Namen als Spalte. Eine für die Highscores, die durch ein Quadrupel aus Nummer, Benutzernamen, Score und Erstellungsdatum verwaltet werden. Zuletzt eine Tabelle für die Freundesliste, die die zwei Befreundeten Benutzernamen speichert (siehe Abbildung 4.27).

Um auf die Datenbank zugreifen zu können, stellt der Server die benötigten *PHP*-Dateien zur Verfügung, die mit der entsprechenden *URL* aufgerufen werden können. Um die gewünschten Daten zu erhalten wird eine *Http-GET* Anfrage mit den benötigten Pa-

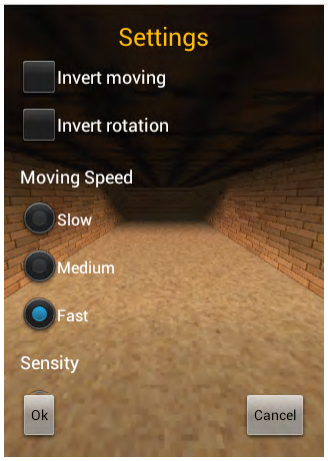

Abbildung 4.25: Activity für die Einstellungen

rameter an die *URL*der erforderlichen *PHP*-Datei gesendet. Der Server bietet hierzu eine Abfrage nach den Highscores an, die Abfrage nach den Highscores der Freunde, nach den Namen der Freunde sowie eine Suche nach den bereits registrierten Benutzern. Anschließend werden die Daten in ein für das mobile Gerät interpretierbares *json*-Objekt geschrieben und zurückgegeben. Um Daten in der Datenbank zu speichern, werden *Http-POST* Nachrichten an die entsprechenden Dateien des Server gesendet und nach erfolgreicher Authentifizierung, die übergebenen Parameter in der Datenbank gespeichert. Somit können ein Benutzer angelegt, ein Score gespeichert sowie die Freundesliste aktualisiert werden. Damit die mobile Applikation mit dem Server kommunizieren kann, werden die *Http-GET* und *Http-POST* Nachrichten in der von Android zur Verügung gestellten Klasse *AsyncTask* gekapselt. Der *AsyncTask* erstellt einen *Thread* sodass die Abfrage im Hintergrund ausgeführt werden kann und erlaubt somit einen asynchronen Aufruf der gewünschten PHP-Datei.

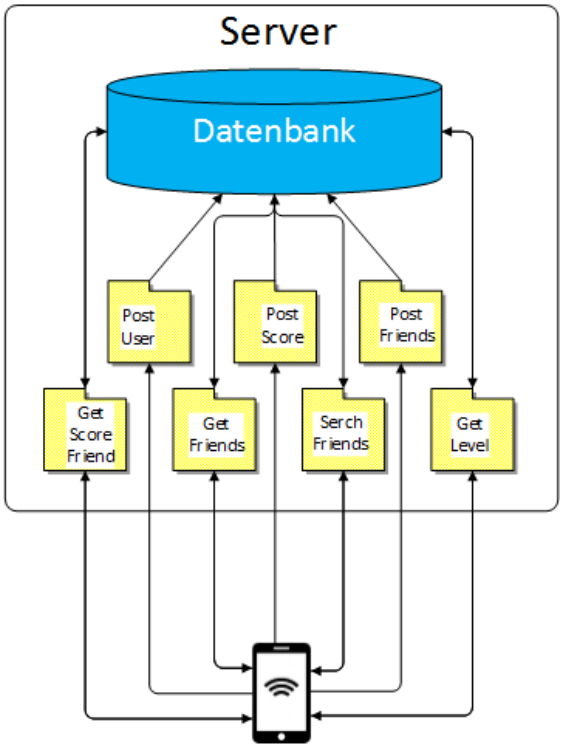

Abbildung 4.26: Datenbankkommunikation

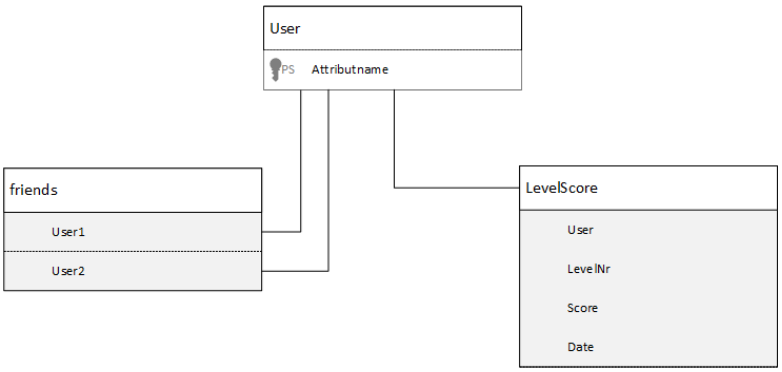

Abbildung 4.27: Datenbank Entitäten

4.3 Das Labyrinth als eigenständiges Spiel

Ein Weitere Möglichkeit dass zu Grunde liegende Labyrinth zu verwenden, ist es, ein eigenständiges Spiel daraus zu entwickeln. Das Ziel eines eigenständigen Spieles ist es, den Spieler früh für das Spiel zu begeistern und den Spielspaß möglichst lange Aufrecht zu erhalten. Um von Anfang an Interesse an dem Spiel zu wecken, wurde dem Spiel daher der reizvolle Name “Just a Maze thing“ gegeben, dass die englischen Wörter “Maze“ (dt.: Labyrinth) und “amazing“ verbindet. Es wird in den folgenden Abschnitten erläutert, wie die Anforderungen dahingehend umgesetzt wurden, um die Ziele des eigenständigen Spieles zu erreichen.

4.3.1 Dialogsystem

Zuerst muss das Spiel in ein Dialogsystem eingebettet werden, damit die angebotenen Funktionen aufgerufen werden können (siehe Abbildung 4.28). Das Spiel startet zunächst in einem Hauptmenü, von dem aus der Spieler sich durch die einzelnen Funktionen navigieren kann. Beim Anordnen der Buttons wurde sich hier für ein Kachelsystem entschieden, da die Anzahl der Buttons relativ niedrig ist und so die Buttons sehr Groß auf der Oberfläche platziert werden können. Dazu wurden die Buttons mit einem Bild und einer Schrift versehen, sodass die Funktionen der Buttons selbsterklärend sind. Das Hauptmenü erlaubt es ein vorher gespieltes Spiel zu laden, eine Levelauswahl zu öffnen, ein zufälliges Spiel zu erstellen, eine Ansicht des Spielfortschritts anzuzeigen, die Einstellungen zu öffnen und das Verlassen des Spieles (siehe Abbildung 4.29.

Abbildung 4.28: Dialogsystem Übersicht

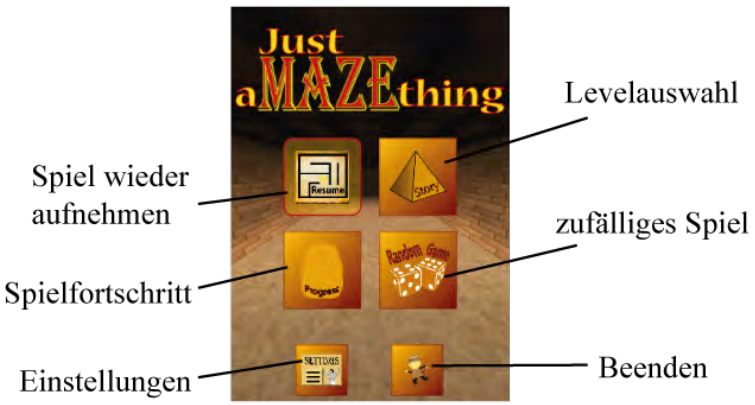

Abbildung 4.29: ansicht des Haptmenüs

4.3.2 Anpassungen des Labyrinthes für das eigenständige Spiel

Für NFA #12 und NFA #13 benötigt es zunächst ein Levelsystem. Man beginnt mit dem ersten Level und mit jedem abgeschlossenen Level schaltet man das nächste Level frei. Zu Beginn sind die Levels noch einfach zu lösen und je fortgeschrittener die Levels, desto höher ist der Schwierigkeitsgrad. Damit soll gewährleistet werden, dass der Spieler schon zu Beginn schnell Erfolgserlebnisse erhält, das Spiel jedoch im späteren Verlauf stets anspruchsvoll bleibt. Somit werden NFA #12 und NFA #13 erfüllt. Analog zu Abschnitt 4.2.2 wird hier der Schwierigkeitsgrad durch die Größe der Level bestimmt. Die Steigung der Levelgröße soll zu Beginn hoch sein und mit höheren Levels abflachen. Es wurde deshalb eine konkave, monoton steigende Funktion gesucht. Durch empirisches Testen wurde die Funktion: $Dim_x = Dim_y := ln(Level * 2) + 4)$ entwickelt. Um weiter die Motivation aufrecht zu erhalten, gibt es zu jedem Level einen Score, der auch zu einem späteren Zeitpunkt verbessert werden kann. Dieser wird nach Abschluss eines jeden Levels berechnet und in die Android internen *SQLite* Datenbank gespeichert. Damit die Level auch später auswählbar sind, und um NFA #14 zu erfüllen, bedarf es einer Oberfläche zur Auswahl der Level, die sich dynamisch erweitert. Es wurde sich, um die Übersichtlichkeit zu erhöhen, dazu entschieden, die absolvierten Level auf mehrere Seiten zu verteilen, die durch Swipegesten navigiert werden (siehe Abbildung 4.30). Dafür sind auf jeder Seite jeweils eine gewisse Anzahl an Level dargestellt, mit deren Nummer, deren erreichten Score sowie ein Symbol um dessen Güte darzustellen. Um dies zu realisieren, wird für jede Seite eine *View* erstellt sowie für jedes Level ein eigenes *RelativeLayout*, das die benötigten Informationen beinhaltet. Die *Views* werden mit der entsprechenden Anzahl an *Levellayouts* befüllt. Wurde die *View* befüllt, wird sie in einer Liste gespeichert und eine weitere *View* wird erstellt. Dies wird solange fortgeführt bis alle bereits gespielten sowie ein weiteres, noch nicht absolviertes Level repräsentiert sind. Die Liste der *Views* wird einem *PagerAdapter* übergeben, der das “durchswipen“ der Seiten realisiert. Somit lassen sich theoretisch unendliche viele Levels frei spielen (NFA #14) und alle zu einem späteren Zeitpunkt wiederholen.

Abbildung 4.30: Activity zur Level Auswahl

Zusätzlich hat der Spieler die Möglichkeit ein Zufällig erstelltes Level zu spielen. Dabei hat der Spieler die Auswahl von fünf verschieden Schwierigkeitsstufen, die wieder durch die Größe der Levels definiert sind (Siehe Abbildung 4.31).

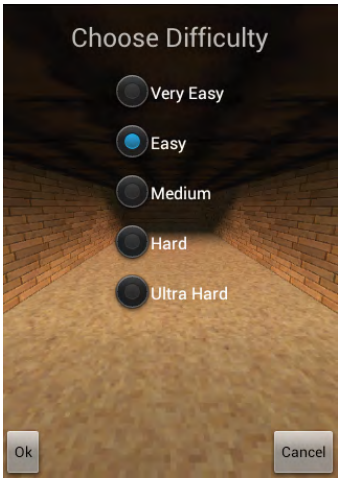

Abbildung 4.31: Activity zur Auswahl eines Zufallsspieles

Des Weiteren hat der Spieler die Möglichkeit, analog zu der *Track Your Tinnitus* Applikation, seinen Fortschritt einzusehen. Allerdings sieht man hier anstatt des Levelranges

den gesamten Score. Außerdem hat der Spieler die Möglichkeit sowohl den Zielpfeil als auch den Kompass auf ewig frei zu schalten. Dies soll zusätzlichen Anreiz geben das Spiel zu spielen und somit zur Erfüllung von NFA #13 dienen. Außerdem zeigte die Erfahrung, dass in späteren Level der Zielpfeil ein notwendiges Utensil ist, um die Übersicht zu behalten. Analog zu Abschnitt 4.2.2 hat der Spieler die Möglichkeit Einstellungen vorzunehmen. Diese werden hier, da der Score hier nicht Zeitbezogen ist, durch eine anpassbare Bewegungsgeschwindikeit erweitert.

5

Abgleich der Anforderungen

Im Folgenden werden die in Kapitel 3 definierten Anforderungen abgeglichen. Dabei wird die Umsetzung der jeweiligen Anforderungen begutachtet und nach dem Schulnotensystem bewertet. Anschließend wird eine Zusammenfassung dieser Bewertung tabellarisch dargestellt.

5.1 Anforderungen an das Labyrinth

Es sollen zunächst die Anforderungen an das grundlegende Spiel bewertet werden. Die einzelnen Anforderungen werden dabei einzeln untersucht.

FA #1 Labyrinth Es wurde eine Bibliothek in die Arbeit eingebettet, die ein Labyrinth mit orthogonaler Tesselation generiert. Damit ist die Anforderung erfüllt. *Bewertung: Sehr gut*

FA #2 Swipegesten Die Bewegungen werden durch Swipegesten durchgeführt. Somit ist diese Anforderung erfüllt.
Bewertung: Sehr gut

FA #3 Schwierigkeitsgrad Der Schwierigkeitsgrad wird in diesem Spiel lediglich über die Veränderung der Labyrinthgröße angepasst. Es wird bei dem Schwierigkeitsgrad jedoch keine Komplexität der Labyrinthstruktur untersucht. Es is möglich, dass ein höheres Level weniger komplex, und somit leichter zu lösen ist, als ein niedrigeres und umgekehrt. Dennoch ist dies im Allgemeinen nicht der Fall und somit wird die Anforderung als erfüllt angesehen.
Bewertung: Ungenügend

FA #4 Levelaufbau Die Level sind nicht vordefiniert und werden automatisch aufgebaut. Ein System das die Texturen in Zahlengruppen unterteilt ermöglicht auch einen abwechslungsreichen Aufbau. Allerdings sehen sich die Levels oftmals sehr ähnlich, und manche Texturen kommen nur äußerst selten zum Vorschein. Dennoch ist die Anforderung erfüllt.
Bewertung: Gut

NFA #5 Licht Die Lichtgestaltung wird durch zwei Arten von Licht realisiert. Das ambiente Licht sorgt für eine Grundhelligkeit und das “Spotlight“ simuliert eine Taschenlampe. Allerdings ruckelt das “Spotlight“. Außerdem wäre es möglich gewesen Materialeigenschaften für die verschiedenen Texturen zu implementieren. Auch hier wäre zwar mehr Potential da gewesen, dennoch trägt das Licht sehr zur Atmosphäre bei.
Bewertung: Gut

FA #6 Items Es gibt drei Arten von Items die den Spieler bei der Wegfindung unterstützen. Außerdem ist der Abruf der Items sehr einfach realisiert. Es sind zwar nicht alle Items sehr hilfreich, allerdings ist das die Ausnahme. Außerdem wären weitere Items denkbar gewesen. Trotzdem wurde die Anforderung erfüllt.
Bewertung: Gut

FA #7 Score Der Score wird im Kontext von *Track Your Tinnitus* anhand der benötigten Zeit sowie der Schrittanzahl des Spielers im Verhältnis zum idealen Zielweg berechnet. Im eigenständigen Spiel wird auf die benötigte Zeit verzichtet. Der Score gibt durchaus die Leistung des Spielers wieder kann für ähnliche Level bei ähnlichem Abschluss aber schwanken. Hier wäre eine stärkere Evaluierung möglich gewesen.
Bewertung: Befriedigend

FA #8 Informationen Während dem laufendem Spiel erscheinen dem Spieler am oberen Rand des Bildschirms Informationen zum aktuellen Spiel. Die Anforderungen wurden hierzu erfüllt, jedoch wird hier auch nichts besonderes angeboten.
Bewertung: Befriedigend

FA #9 Spielwährung Durch aufsammelbare Goldbarren wird eine Spielwährung erworben, mit der es möglich ist Items zu kaufen. Die Anforderung wurde somit erfüllt. Allerdings wären weitere Optionen zum Erhalten der Spielwährung denkbar gewesen.
Bewertung: Gut

FA #10 Map Es ist möglich eine Kartenansicht des Labyrinthes mit wenigen Klicks zu erhalten. Die Karte gliedert sich gut in den Kontext eines Schatzsuchers ein. Außerdem ist sie sehr hilfreich.
Bewertung: Sehr gut

FA # 11 Kompass Es wurde ein Kompass implementiert, der immer im Blickfeld des Spielers ist und der stets nach Norden zeigt. Somit ist die Anforderung erfüllt. Allerdings

wirkt der Kompass unecht und scheint durch das Level zu gleiten. Außerdem ist er lediglich eine Scheibe und schwer lesbar.
Bewertung: Ausreichend

FA #12 Zielpfeil Es gibt einen freischaltbaren Zielpfeil. Dieser zeigt stets in die Richtung des Zieles und liegt stets im Blickfeld des Spielers. Zusätzlich ist er sehr hilfreich bei der Wegfindung. Allerdings gliedert er sich nicht gut in das Erscheinungsbild des Spieles ein.
Bewertung: Gut

NFA #1 Rechteckiges Labyrinth Die Klasse zur Generierung der Labyrinthe ergibt stets ein Rechteck. Damit ist die Anforderung erfüllt.
Bewertung: Sehr gut

NFA #2 Vollständiges Labyrinth Die Klasse *SimpleGridMaze* erstellt ein vollständigen Graphen als Grundlage des Labyrinthes. Dadurch ist jeder Punkt des Labyrinthes begehbar und auch von jedem Punkt aus zu erreichen. Damit ist die Anforderung erfüllt.
Bewertung: Sehr gut

NFA #3 Zielweg Da der Zielweg die zwei entferntesten Punkte eines nicht zyklischen vollständigen Graphen sind, existiert nur ein Zielweg. Damit ist die Anforderung erfüllt.
Bewertung: Sehr gut

NFA #4 Spielumgebung Das Labyrinth wird in einer dreidimensionalen Spielumgebung eingebettet. Damit ist die Anforderung erfüllt.
Bewertung: Sehr gut

NFA #5 Bedienung Die Bewegungen werden durch einfache Swipegesten gesteuert. Eine einfache Geste steuert eine ganze Bewegung an. Dadurch sind die Bewegungen schnell durchzuführen. Die Intuitivität der Steuerung ist abhängig von den Gewohnheiten

des Benutzers. Durch die Möglichkeit die Steuerung zu invertieren, kann darauf jedoch eingegangen werden. Somit kann von einer intuitiven Steuerung gesprochen werden. Ebenso lässt sich die Sensitivität einstellen. Somit ist auch eine präzise Steuerung gegeben, die die gewollten Bewegungen des Benutzers präzise ausführt.
Bewertung: Sehr gut

NFA #6 Levelart Es gibt sowohl verschieden große Levels als auch unterschiedlich komplexe. Da die Komplexität nicht bestimmt wird, sondern zufällig geschieht, muss die Betrachtung der Komplexität dennoch als nicht erfüllt angesehen werden.
Bewertung: Ausreichend

NFA #7 Atmosphäre Das Spiel taucht in den Kontext einer ägyptischen Pyramide. Kontextbezogene Texturen erlauben diesen Eindruck und Totenschädel und Skelette verstärken diesen. Ebenso gibt das Licht den Eindruck einer Taschenlampe, sodass die Atmosphäre eines Schatzsuchers aufkommt. Allerdings sehen die Texturen oft unecht aus. Verschiedene Texturen haben keinen laufenden Übergang und wirken aufgesetzt. Dazu erscheint das Licht nicht realistisch.
Bewertung: Befriedigend

NFA #9 Labyrinthgestaltung Durch verschiedene Texturen, die im Level verteilt werden, erscheint dass Level auf den ersten Blick sehr abwechslungsreich. Es könnte dennoch wesentlich mehr Texturen miteinbezogen werden und es sind auch weitere Objekte, die in der Welt verteilt werden, denkbar. Außerdem wäre es möglich gewesen das Spiel mit zusätzlichen Textursets zu ergänzen und damit die Level unterschiedlich zu gestalten. Die Anforderung ist erfüllt, jedoch ist hier mehr Potential vorhanden.
Bewertung: Befriedigend

NFA #10 Performance Durch einen optimierten Levelaufbau konnte die dafür benötigte Zeit erheblich reduziert werden. Außerdem wurde dadurch die CPU und GPU

Auslastung während der Laufzeit gemindert. Dennoch erfordert vor allem die Lichtgestaltung viele Ressourcen. Dieser Kompromiss wurde jedoch für eine bessere Atmosphäre geschlossen. *Bewertung: Gut*

5.2 Anforderungen im Kontext *Track Your Tinnitus*

Im folgenden Abschnitt wird nun die Realisierung der Anforderungen an das Spiel im Kontext *Track Your Tinnitus* betrachtet. Anschließend werden die Anforderungen abgeglichen und ein Bewertung erstellt.

FA #13 Aufmerksamkeit Das Finden des Zielweges erfordert vom Spieler Entscheidungen über die Wahl der richtigen Abzweigung, das die Aufmerksamkeit des Spielers erfordert. Das Ziel ist allerdings durch die Beschränkung der Spielzeit und somit der Größe der Levels auch durch triviales Ablaufen der Gänge möglich. Dennoch lässt die stimmige Atmosphäre den Spieler in die Welt eintauchen und die einfache Steuerung hebt den Spaß durch das Level zu schreiten, was die Aufmerksamkeit des Spielers fördert.
Bewertung: Befriedigend

FA #14 Schwierigkeitsgrad Der Schwierigkeitsgrad gleicht sich automatisch den Fähigkeiten des Spielers an. Da auch hier wieder das Problem gegeben, dass keine Aussage oder Anpassung der Komplexität der Level möglich ist kann die Anforderung nicht als erfüllt betrachtet werden.
Bewertung: ungenügend

FA #15 Scorekategorisierung Der errechnete Score wird durch eine sehr gute und überlegte Funktion kategorisiert und erlaubt somit eine gute Approximation an die gewünschte Spieldauer. Damit ist die Anforderung erfüllt.
Bewertung: Sehr gut

FA #16 Bestenliste Es wurde eine Bestenliste implementiert, die dem Benutzer einen Vergleich mit anderen Spielern und Freunden ermöglicht. Auf Grund der Vielzahl an verschiedenen Levels werden diese allerdings anhand der Länge des Zielweges und somit nicht identische Levels miteinander verglichen. Auch ist die Ansicht der Bestenliste unübersichtlich und die Auswahl der Level schwierig.
Bewertung: Ausreichend

FA #17 Freundesliste Es ist möglich Freunde zu suchen, eine Freundesliste zu erstellen und diese zu verwalten. Dies ist alles sehr einfach und übersichtlich gehalten. Lediglich das Design hat Verbesserungspotential.
Bewertung: Gut

FA #18 Server Es existiert eine Serververbindung, die die Bestenlisten, Benutzer sowie die Freundeslisten verwaltet. Die Applikation reagiert sehr robust auf eine fehlende Serververbindung, allerdings gibt sie wenig Feedback über die Ursache. Ebenso wurden Aspekte der Serversicherheit vernachlässigt. Daher erfüllt sie den minimalen Bedingungen.
Bewertung: Befriedigend

NFA #11 Spieldauer Aufgrund einer guten Kategorisierung des Scores könnte die Spieldauer im Allgemeinen gut gegen die geforderte Zeit approximieren. Die Variation der Komplexität der Level wirkt dieser Approximation allerdings entgegen, da ein Level höheren Ranges weniger komplex und somit schneller lösbar sein kann, als ein niederrangiges Level, und umgekehrt.
Bewertung: ausreichend

5.3 Anforderungen an das eigenständige Spiel

In diesem Abschnitt werden die Anforderungen an das eigenständige Spiel abgeglichen. Eine Bewertung stuft diese anschließend ein.

FA #18 Schwierigkeitsgrad Durch die steigende Levelgröße steigert sich im Allgemeinen auch der Schwierigkeitsgrad. Die Tatsache dass das in vereinzelten Fällen nicht zutrifft, ist hier vernachlässigbar.
Bewertung:Gut

NFA #12 Einsteigerfreundlichkeit Aufgrund der sehr leichtem Level zu Beginn und der intuitiven und schnell erlernbaren Steuerung erhält der Spieler früh Erfolgserlebnisse, das zur Einsteigerfreundlichkeit beiträgt. Eine kontextbezogene Hintergrundgeschichte sowie eine Einführung in die Steuerung könnten den Einstieg jedoch verbessern.
Bewertung: Gut

NFA #13 Langzeitmotivation Durch die Endlose Zahl an Levels und der Ungewissheit ob dieses Spiel irgendwann zu Ende ist,lassen den Spieler stets weiter Spielen. Mit dem steigendem Schwierigkeitsgrad bleibt das Spiel dabei anspruchsvoll. Ebenso geben die auch auf ewig freischaltbaren Items weiterhin Anreizdas Spiel fortzusetzen. Besonders der Zielpfeil gibt nochmal ein neues Spielgefühl im späteren Spielverlauf. Die Statistik wertet das Spiel im längerem Spielverlauf zusätzlich auf. Trotzdem wirken die immer gleichen Leveldesigns auf Dauer eintönig. Weitere Textursets hätten hier die Optik auf längere Zeit interssant gehalten.
Bewertung: Gut

NFA #14 Levelanzahl Die Anzahl der Levels ist rein theoretisch unbeschränkt. Praktisch gesehen begrenzt jedoch die Berechnung des Levelaufbaus Anzahl. So dauert der Levelaufbau bei sehr hohen Levels abhängig vom verwendetem Gerät bereits mehrere Sekunden.
Bewertung: Gut

5.4 Zusammenfassung

In diesem Abschnitt werden die Anforderung gemeinsam mit deren Bewertungen nochmals tabellarisch Zusammengefasst.

Tabelle 5.1: Tabelle der allgemeinen Anforederungen

Note	1	2	3	4	5	6
FA #1 Labyrinth	✓					
FA #2 Swipegesten	✓					
FA #3 Schwierigkeitsgrad					✓	
FA #4 Levelaufbau		✓				
FA #5 Licht		✓				
FA #6 Items		✓				
FA #7 Score			✓			
FA #8 Informationen			✓			
FA #9 Spielwährung		✓				
FA #10 Map	✓					
FA #11 Kompass				✓		
FA #12 Zielpfeil		✓				
NFA #1 Rechteckiges Labyrinth	✓					
NFA #2 Vollständiges Labyrinth	✓					
NFA #3 Zielweg	✓					
NFA #4 Spielumgebung	✓					
NFA #5 Bedienung	✓					
NFA #6 Levelart				✓		
NFA #7 Atmosphäre			✓			
NFA #9 Labyrinthgestaltung			✓			
FA #10 Performance		✓				

Tabelle 5.2: Tabelle der Anforderungen im Kontext *Track Your Tinnitus*

Note	1	2	3	4	5	6
FA #13 Aufmerksamkeit			✓			
FA #14 Schwierigkeitsgrad					✓	
FA #15 Scorekategorisierung	✓					
FA #16 Bestenliste				✓		
FA #17 Freundesliste		✓				
FA #18 Server			✓			
NFA #11 Spieldauer				✓		

Tabelle 5.3: Tabelle der Anforderungen an das eigenständige Spiel

Note	1	2	3	4	5	6
FA #18 Schwierigkeitsgrad		✓				
NFA #12 Einsteigerfreundlichkeit		✓				
NFA #13 Langzeitmotivation		✓				
NFA #14 Levelanzahl		✓				

6

Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse der Arbeit zunächst zusammengefasst und kritisch Betrachtet. Abschließend wird ein Ausblick für Erweiterungen und späteren Nutzen der erarbeiteten Konzepte dieser Arbeit gegeben sowie auf einen möglichen Verwendungsbereich der abgeschlossenen Entwicklung eingegangen.

6.1 Zusammenfassung

Im Rahmen dieser Arbeit ist ein Spiel entstanden, das ein einfaches zweidimensionales Labyrinth erstellt und dieses mittels der Grafikbibliothek OpenGL in einer dreidimensionalen Umgebung abbildet. Im Zuge der Entwicklung der Arbeit wurde dabei ein Verständnis für die Grundsätze von OpenGL erlangt. Die Arbeit zeigt einen Überblick der Architektur und der Arbeitsweise von OpenGL sowie ein zielorientiertes Anwen-

den bestimmter Aspekte. Dabei konnte allerdings kein tiefergehender Einblick in die Kernberechnungen und Funktionsausführungen von OpenGL gegeben werden. Ebenso wurde die *Rendering-Pipeline* abstrakt betrachtet. Dies musste aufgrund der Tiefe dieser Thematik vernachlässigt werden. Hier wäre eine weiterführende Forschungsarbeit denkbar. Zusätzlich wurden Grundlagen der Labyrinththeorie dargestellt und in diesem Zusammenhang ein Algorithmus zur Generierung eines Labyrinthes erörtert. Ein Vergleich von verschiedenen Algorithmen wurde dabei nicht betrachtet. Diese Grundlagen wurden anschließend in der Realisierung des Spieles berücksichtigt. Dabei hat sich die Trennung zwischen der Einbettung in die Android-Programmierung und der generellen Programmierung der Spiellogik als sehr übersichtlich herausgestellt. Des Weiteren konnte eine gute Modularisierung zwischen dem Levelaufbau, der Positionskontrolle und der Bewegungskontrolle erarbeitet werden. Diese Modularisierung erlaubte es im späteren Stadium der Entwicklung den Levelaufbau dahingehend zu gestalten, dass eine bessere Performance erreicht werden konnte. Das entwickelte Spiel konnte anschließend als Basis für eine Erweiterung der *Track Your Tinnitus* Applikation genutzt werden. Dazu wurde das Symptom Tinnitus erklärt und dessen Behandlung mit der Applikation *Track Your Tinnitus* dargestellt. Dabei konnten nicht alle Anforderungen erfüllt werden. Dies lässt sich auf einen fehlenden Vergleich der Algorithmen für die Labyrinthgenerierung zurückführen. Trotzdem konnte das Ziel, den Spieler von seinem Tinnitus abzulenken, hinreichend erfüllt werden. Gründe dafür sind die authentische Spielumgebung sowie der gelungene Bewegungsapparat. Diese Aspekte dienen auch der Erfüllung der Anforderungen an das eigenständige Spiel.

6.2 Ausblick

Diese Arbeit hat ein Konzept verwendet, das die Trennung der Spiellogik von der Einbettung in das Betriebssystem Android verlangte. Da sich diese Trennung als sehr übersichtlich erwiesen hat, ist dieses Prinzip für weitere Anwendungen zu empfehlen und kann auch plattformübergreifend verwendet werden. Ebenso ist die Modularisierung des Spieles für eine zukünftige Verwendung hilfreich. Sie erlaubt es die Labyrinthgenerierung, den Levelaufbau, die Positionskontrolle oder die Bewegungskontrolle aus-

zutauschen und dadurch ein neues Spielgefühl zu vermitteln. Damit die Anforderungen bezüglich der Erweiterung der *Track Your Tinnitus* später erfüllt werden können, kann somit die zugrunde liegende Bibliothek für die Labyrinthgenerierung ersetzt werden. Ebenso könnte das Spiel als Basis für viele weitere Variationen dienen. Es ist vorstellbar aus dem Spiel ein dreidimensionales Labyrinth zu entwickeln, indem man mehrere zweidimensionale Labyrinthe übereinander legt und sie über Wege nach oben und unten miteinander verbindet. Man stelle sich nun diese Stapelung zu einem Zauberwürfel vor. Durch zeitbedingtes Drehen der Ränder könnte der Rätselfaktor zusätzlich erhöht werden. Man könnte auch das Spiel mit internen Rätsel erweitern, die gelöst werden müssen, bevor das Level beendet werden kann. Ein weiteres Szenario wäre das Spiel als Grundlage für ein minimalistisches Rollenspiel zu nehmen. Auftauchende Monster müssten durch einfache Angriffe besiegt werden wofür der Spieler Erfahrungspunkte erhält. Doch die wohl interessanteste Erweiterung wäre, das Brettspiel “Das verrückte Labyrinth“ auf Grundlage des entstandenen Spieles zu realisieren.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Bun14a] Bundesverband Interaktive Unterhaltungssoftware e.V. Boom im digitalen Spielemarkt. http://www.biu-online.de/de/presse/newsroom/newsroom-detail/datum/2014/01/21/boom-im-digitalen-spielemarkt.html, 2014. [Zuletzt besucht am 15.08.2014].

[Bun14b] Bundesverband Interaktive Unterhaltungssoftware e.V. Spiele-Apps beliebt wie nie. http://www.biu-online.de/de/presse/newsroom/newsroom-detail/datum/2014/09/01/spiele-apps-beliebt-wie-nie.html, 2014. [Zuletzt besucht am 15.08.2014].

[CGT13] CGTextures. Cgtextures license. http://www.cgtextures.com, 2013. [Zuletzt besucht am 05.10.2014].

[Cla97] Ute Claussen. *Programmieren mit OpenGL: 3D-Grafik und Bildverarbeitung*. Springer, 1997.

[Fol08] Martin Foltin. Automated maze generation and human interaction. 2008.

[GSP+14] Philip Geiger, Marc Schickler, Rüdiger Pryss, Johannes Schobel, and Manfred Reichert. Location-based mobile augmented reality applications: Challenges, examples, lessons learned. In *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, pages 383–394, April 2014.

[HA01] Kevin Hawkins and Dave Astle. *OpenGL game programming*. Thomson Course Technology, 2001.

[Hit02] Ron Hitchens. *Java NIO*. OReilly Media, Inc., 2002.

[Inc14] Google Inc. *Android Developers API*. http://developer.android.com/reference/packages.html, 2014. [Zuletzt besucht am 25.10.2014].

[Ini13] Tinnitus Research Initiative. Track Your Tinnitus. https://www.trackyourtinnitus.org, 2013. [Zuletzt besucht am 03.09.2014].

[Lin14] Michael Lindinger. *Konzeption und Implementierung einer mobilen Anwendung zur Unterstützung von Tinnitus-Patienten*. University of Ulm, 2014.

[Ope12] OpenGL. *OpenGL ES Documentation*. https://www.opengl.org/sdk/docs/man, 2012. [Zuletzt besucht am 05.10.2014].

[SHP+14a] Winfried Schlee, Jochen Herrmann, Rüdiger Pryss, Manfred Reichert, and Berthold Langguth. How dynamic is the continuous tinnitus percept? In *11th International Tinnitus Seminar*, May 2014.

[SHP+14b] Winfried Schlee, Jochen Herrmann, Rüdiger Pryss, Manfred Reichert, and Berthold Langguth. Moment-to-moment variability of the auditory phantom perception in chronic tinnitus. In *13th Int'l Conf on Cochlear Implants and Other Implantable Auditory Technologies*, June 2014.

[SRLP+13] Johannes Schobel, Martina Ruf-Leuschner, Rüdiger Pryss, Manfred Reichert, Marc Schickler, Maggie Schauer, Roland Weierstall, Dorothea Isele, Corina Nandi, and Thomas Elbert. A generic questionnaire framework supporting psychological studies with smartphone technologies. In *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, pages 69–69, June 2013.

[SSP+13] Johannes Schobel, Marc Schickler, Rüdiger Pryss, Hans Nienhaus, and Manfred Reichert. Using vital sensors in mobile healthcare business applications: Challenges, examples, lessons learned. In *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, pages 509–518, May 2013.

[SSP+14] Johannes Schobel, Marc Schickler, Rüdiger Pryss, Fabian Maier, and Manfred Reichert. Towards process-driven mobile data collection applications:

Requirements, challenges, lessons learned. In *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, pages 371–382, April 2014.

[Tur09] Volker Turau. *Algorithmische Graphentheorie*. Oldenbourg Verlag, 2009.

[Zec11] Mario Zechner. *Beginning Android Games*. Springer, 2011.

Name: Julian Schweiger Matrikelnummer: 711784

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Julian Schweiger