IRVANIA

3. Abgabe

CG2LU, SS 2007
REITERER Martin, 532, 0525939, e0525939@student.tuwien.ac.at
FRITSCH Barbara, 532, 0525947, e0525947@student.tuwien.ac.at

Inhaltsverzeichnis

1 Implementierung der Anforderungen

In diesem Kapitel beschreiben wir, wie wir die an uns gestellten Anforderungen in IRVANIA umgesetzt haben.

1.1 Gameplay

Bei unserem Spiel handelt es sich um die digitale Nachbildung eines Brettspiels. Hierbei besitzt der Spieler die Kontrolle über ein Spielbrett, welches er über sämtliche Achsen drehen kann. Zu Spielbeginn wird eine Kugel in die Welt bzw. auf das Brett geworfen, die durch Neigen des Brettes in ein deutlich markiertes Ziel bewegt werden muss. Die Kugel rollt dabei immer in die aktuelle Neigungsrichtung der Bodenfläche.

Ziel des Spiels ist es alle Artefakte einzusammeln um den Ausgang frei zuschalten. Um es etwas komplizierter zu machen, darf man nur drei Mal vom Spielbrett runterfallen. Fällt man öfters hinunter ist das Spiel vorbei und man muss von vorne beginnen.

Die in der 2. Abgabe enthalten physikalischen Berechnungen haben wir in dieser Abgabe durch PhysX ersetzt. Hierfür halten wir in unserer internen Datenstruktur neben den Geometrieinformationen zusätzliche PhysX-Actors für folgende 3D-Objekte, welche miteinander interagieren können:

3D-Objekt	PhysX-Actor
Kugel	Sphere-Shape-Actor
Spielbrett	Box-Shape-Actor
Wand	Box-Shape-Actor
Ausgang	Box-Shape-Actor
Blocker	Box-Shape-Actor

Um eine Verbindung zwischen Geometrie und PhysX-Objekten herzustellen, haben wir jeden Actor eindeutig über die User-Data identifiziert. Jeder Actor besitzt folgende Attribute:

- *Name*
  eindeutig identifizierender Bezeichner

- *Collision-Flag*
  Gibt an, ob mit diesem Objekt eine Berührung aufgetreten ist. Diese Flag wird in der Collision-Callback-Function aktualisiert.

1.2 Nicht triviale / animierte Objekte

Neben dem bewusst trivial gehaltenen Spielbrett gibt es in IRVANIA auch einige nicht triviale Objekte. Diese sind zum Beispiel im ersten Level ein Gartentisch sowie im zweiten Level eine Windmühle. Weiters handelt es sich bei diesen Elementen um animierte 3D-Objekte. Es folgt eine kurze Beschreibung dieser Objekte:

Gartentisch
Der Gartentisch besteht aus zwei fbx Dateien. In tisch.fbx sind die statischen Elemente, wie der Tisch, die Beine und alles was auf dem Tisch ist, abgespeichert. Zusätzlich gibt es noch eine weitere fbx Datei, die sonnenschirm.fbx heißt, in dieser befindet sich der Sonnenschirm und die Stange, die animiert werden. Der Sonnenschirm rotiert entlang der y-Achse und der z-Achse. Wobei die z-Achse nur im Intervall -45° und 0° liegen darf, damit der Sonnenschirm nicht durch den Boden rotiert.

Windmühle
Analog zum Gartentisch, ist die Windmühle ebenfalls in eine statische und eine dynamische fbx Datei aufgeteilt. Die dynamischen Elemente sind hier die Blätter und das Verbindungsstück zur Windmühle, der Rest ist statisch. Die Animation der Blätter erfolgt entlang der x-Achse und hat keine Intervalleinschränkung.

Blocker
Der Blocker ist zwar ein triviales Objekt, wird aber animiert wenn alle Artefakte eingesammelt wurden. Er rotiert um die y-Achse und wird dabei gleichzeitig so lange skaliert, bis dessen Größe 0 ist. Dabei wandert er entlang der y-Achse nach oben.

1.3 Beschleunigung der Sichtbarkeitsberechnung

Mit der Implementierung des Frustum Cullings wurde bereits begonnen. Es gibt eine Klasse ViewFrustum in der die 6 Frustum-Planes berechnet werden und eine Methode, die überprüft, ob der Punkt in der BoundingSphere liegt. Es fehlt uns der Aufruf der Methode, sodass überprüft werden kann, welche Teile des Hintergrundes angezeigt werden sollen. Wir haben uns überlegt, dass wir den Hintergrund splitten und dann bestimmte Rechtecke einfach nicht rendern.

1.4 Transparenz-Effekte

Um transparente Objekte darstellen zu können, haben wird die Blending Funktion durch den Befehl `glBlendFunc (GL_SRC_ALPHA, GL_DST_ALPHA);` eingeschaltet bzw. `glBlendFunc(GL_ONE, GL_ZERO);` wieder ausgeschaltet.

1.5 Vertex-Arrays / Immediate Mode / VBO

OpenGL besitzt einige verschiedene Arten um Geometriedaten aufzunehmen. Diese unterscheiden sich jedoch sehr stark in ihrer Performance, so sind zB. Vertex-Buffer-Objects die schnellste Variante um Koordinaten in den Grafikspeicher zu laden. Im Gegensatz zu VBO bildet der Immediate-Mode die mit Abstand schlechteste Performance, da sich OpenGL erst aus den einzelnen Daten eine vollständige Beschreibung der Geometrie ableiten muss.

Immediate Mode
Im Immediate Mode wird jede einzelne Koordinate über einen eigenen Function-Call an OpenGL übergeben. Dies kann bei großen Objekten zu einem Flaschenhals führen, da Function-Calls auf einigen Rechnern zu einem enormen Overhead führen.

Codeausschnitt für das Rendern eines 3D-Objektes im Immediate Mode:

```
void Object3D::setVertexImmediate (int iObject)
{
   glBegin( GL_TRIANGLES );

   int j = 0;
   for (int i = 0; i <= numVertices[iObject] *3; i+=3){
      glNormal3f( normals[iObject][i], normals[iObject][i+1], normals[iObject][i+2]);
      glMultiTexCoord2f( GL_TEXTURE1, textureUV[iObject][j], textureUV[iObject][j+1] );
      glVertex3f( vertices[iObject][i], vertices[iObject][i+1], vertices[iObject][i+2]);
      j += 2;
   }

   glEnd ();
}
```

Vertex-Arrays

Um nicht jede Koordinate einzeln an OpenGL übergeben zu müssen, gibt es auch die Möglichkeit alle Punkte/Koordinaten in einem einzigen Function-Call dem Hauptspeicher als Array zu übergeben. Das hat den Vorteil, dass nicht alle einzelnen Koordinaten, die zu einem Geometrieobjekt gehören, mühsam zusammengesucht werden müssen.

Codeausschnitt für das Rendern eines 3D-Objektes mittels Vertex-Arrays:

```
void Object3D::setVertexArrays(int iObject)
{
   /* Vertex-Array in den Client-Speicher laden */
   glVertexPointer (3, GL_FLOAT, 0,vertices[iObject]);
   /* Normalvektoren-Array in den Client-Speicher laden */
   glNormalPointer (GL_FLOAT, 0, normals[iObject]);
   /* Texturkoordinaten in den Clientspeicher laden */
   glTexCoordPointer(2, GL_FLOAT, 0, textureUV[iObject]);

   /* Objekt zeichnen */
   glDrawArrays(GL_TRIANGLES,0,numVertices[iObject]);
}
```

Der Source-Code zeigt bereits, dass es sich um eine elegantere Lösung handelt.

Vertex-Buffer-Objects (VBO)

Man kann eine noch größere Verbesserung der Performance erhalten indem man die einzelnen Geometrieinformationen direkt in den Grafikspeicher der Grafikkarte lädt. Dies ist durch VBO möglich. Die Funktionalität ist ähnlich zur Verwendung von Vertex-Arrays, mit dem Unterschied, dass die Arrays, welche geladen werden, sich nicht im Hauptspeicher sondern im Grafikspeicher befinden.

Codeausschnitt zum Rendern eines 3D-Objektes mittels VBO:

```
void Object3D::setVertexBuffer (int iObject)
{
   if (!bufferInit)
   {
      if (!bufferInit)
         vBuffer[iObject] = new GLuint[3];
      /* VERTEX-BUFFER */
      glGenBuffersARB ( 1, &vBuffer[iObject][0]);
      glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vBuffer[iObject][0] );
      glBufferDataARB ( GL_ARRAY_BUFFER_ARB, numVertices[iObject] * 3 * sizeof(GLfloat),
                  vertices[iObject], GL_STATIC_DRAW);
      /* NORMAL-BUFFER */
      glGenBuffersARB ( 1, &vBuffer[iObject][1]);
      glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vBuffer[iObject][1] );
      glBufferDataARB ( GL_ARRAY_BUFFER_ARB, numVertices[iObject] * 3 * sizeof(GLfloat),
```

```
                    normals[iObject], GL_STATIC_DRAW);
        /* TEXTURE-BUFFER */
        glGenBuffersARB ( 1, &vBuffer[iObject][2]);
        glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vBuffer[iObject][2] );
        glBufferDataARB ( GL_ARRAY_BUFFER_ARB, numVertices[iObject] * 2 * sizeof(GLfloat),
           textureUV[iObject], GL_STATIC_DRAW);
    }

    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vBuffer[iObject][0] );
    glVertexPointer ( 3, GL_FLOAT, 0, (char *) NULL);
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vBuffer[iObject][1] );
    glNormalPointer ( GL_FLOAT, 0, (char *) NULL);
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vBuffer[iObject][2] );
    glTexCoordPointer ( 2, GL_FLOAT, 0, (char *) NULL);

    glDrawArrays ( GL_TRIANGLES, 0, numVertices[iObject] );
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, 0 );
}
```

Die oben angeführte Render-Methode zeigt das Rendern von 3D-Objekten mithilfe von VBO. Hierbei wird die Geometrie beim aller ersten Render-Durchlauf in den Vertex-Buffer geschrieben. In den darauf folgenden Render-Durchläufen bedient sich diese Funktion mit den bereits im Speicher liegenden Objektdaten.

1.6 Display-Listen

Wir haben eine Displayliste, die den Hintergrund (linke, rechte, hintere, vordere, obere und untere Wand) beinhaltet. Diese Displayliste enthält alle Transformationen, da sich die Werte nicht mehr ändern. Für alle dynamischen Objekte haben wir eine eigene Displayliste erstellt, die immer in der render Funktion statt dem „normalen" render ausgeführt werden. Die Berechnungen der dynamischen Objekte müssen vor dem glCallList() erfolgen, da die Displayliste keine veränderten Werte berücksichtigt.

1.7 Steuerung

Die Steuerung funktioniert hauptsächlich über die Maus. Durch Bewegung dieser, kann der Spieler das Spielbrett in die gewünschte Richtung neigen. Möchte man bei neutraler Position des Spielbrettes, die Kugel nach vor rollen lassen, so muss man lediglich die Maus nach vorne bewegen. Nun neigt sich das Spielbrett nach vorne, was dazuführt, dass auch die Kugel in diese Richtung rollt.

Die Steuerung erfolgt hierbei über die Extrahierung der Neigungsänderungen aus den verschiedenen Mausbewegungen. Die daraus resultierenden Rotationsmatrizen werden im Anschluss auf die intern gehaltenen PhysX-Actors des Spielbretts und der Kugel angewandt.

Weiters kann der Benutzer die aktuelle Blickrichtung auf das Spielfeld verändern, indem er durch drücken der LEERTASTE/LINKEN MAUSTASTE in den Pause-Modus wechselt. Hier wird die Kamera um den Mittelpunkt nach links/rechts rotiert, wenn sich die Maus dementsprechend bewegt.

Erweiterte Steuerung der Funktionen:

Shortcut	**Funktion**
F1	Hilfe
F / F2	Framerateanzeige ein-/ausschalten

F3	Wire Frame ein-/ausschalten
F4	Texturqualität verändern: Nearest Neighbor / Bilinear
F5	MipMapping-Qualität ändern: Aus / Nearest Neighbor / Linear
F6	Umschalten zw.: Vertex-Arrays, Immediate Mode und VBO
F7	Display-Listen ein-/ausschalten
F8	View frustum culling ein-/ausschalten
F9	Transparenz ein-/ausschalten
S	direkt zum zweiten Level springen

2 Spezialeffekte

Im folgenden Abschnitt wollen wir näher auf die von uns implementierten Spezialeffekte eingehen.

2.1 Shadow Mapping

Wie bereits in der 2. Abgabe erwähnt haben wir uns entschieden Schatten zu erzeugen, da dieser Effekt in unserem Spiel sehr gut zu sehen ist. Weiters ist es dem Spieler so besser möglich Tiefenunterschiede zu erkennen.

Da wir zuvor fast keine Ahnung über Schatten hatten, haben wir uns auf folgende Informationsquellen gestützt:

- *Paul's Projects*
  http://www.paulsprojects.net/tutorials/smt/smt.html

  Gut für allgemeine Informationen über Shadow-Mapping und die Berechnung der Textur-Matrix.
  Aber eher schlecht für das Rendering-Verfahren selbst, da drei Durchläufe benötigt werden. Der Grund dafür ist, dass in diesem Tutorial der Schatten nicht durch Shader entsteht, sondern durch überblenden eines Schattenbildes mit dem normalen Bild.

- *Orange & Red Book*
  für das Laden von Shadern bzw. für Basics in Bezug zu GLSL.

- *Informatik-Forum*
  http://www.informatik-forum.at/showthread.php?t=55089

  Wenn uns die Ideen ausgegangen sind, gab es hier immer hilfreiche Unterstützung. Danke an dieser Stelle!

- *Delphi-GL-Wiki*
  http://wiki.delphigl.com/index.php/GL_EXT_framebuffer_object

  Hier findet man sehr gute Beschreibungen zu OpenGL-Konzepten. Unter anderem haben wir von hier auch den Überblick über Frame-Buffer-Objects erhalten.

Zur Implementierung
Die Schatten werden bei uns im Gegensatz zum Tutorial von Paul's Projects in zwei Schritten erzeugt:

1. Wir bewegen die Kamera an die Position der Lichtquelle und rendern den Tiefenbuffer in ein Frame-Buffer-Object der Größe von 1024x1024 Pixel.

2. Nun berechnen wir die Texturmatrix und übergeben das zuvor mit der Shadow Map initialisierte FBO an unseren „Shadow Map" Shader.

Die eigentliche Berechnung der Schatten geschieht im Shader. Hier berechnet uns der Vertex-Shader einerseits die Texturkoordinaten für die Shadow-Map (Texturebene 0) und andererseits die Texturkoordinaten für die normale Textur (Texturebene 1). Im Fragment Shader wird nun die normale Texturierung berechnet und überprüft, ob sich das aktuelle

Fragment im Schatten befindet. Ist das Fragment im Schatten, so wird es entsprechend dunkler angezeigt.
Weiters haben wir die aufgrund der begrenzten Auflösung der Shadow Map kantigen Schattenränder mittels Antialiasing geglättet.

Ausschnitt aus dem Fragmentshader für das Überprüfen, ob ein Fragment im Schatten liegt:

```
float depth = shadow2DProj(shadowMap, projCoord + vec4(x, y,0.0,0.0) * 0.03).r;
```

2.2 Phong-Shading

Da die Kugel sowie die Artefakte glänzende Oberflächen haben sollten und wir ohnehin einen Shader-Loader für Shadow Mapping implementiert haben, haben wir uns entschieden auch einen Phong-Shader zu schreiben.

Sämtliche Berechnungen haben wir so wie in der CG1 Vorlesung gezeigt durchgeführt.

Codeausschnitt für die Berechnung der Illiuminace eines Fragments:

```
varying vec3 normal;
varying vec3 v;
varying vec3 lightvec;

void main(void)
{
  vec3 Eye = normalize(-v);

  vec3 Reflected = normalize( reflect( -lightvec, normal ));

  vec4 IAmbient = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
  vec4 IDiffuse = gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse * max(dot(normal,
lightvec), 0.0);
  vec4 ISpecular = gl_LightSource[0].specular * gl_FrontMaterial.specular *
pow(max(dot(Reflected, Eye), 0.0), gl_FrontMaterial.shininess);
  gl_FragColor = gl_FrontLightModelProduct.sceneColor + IAmbient + IDiffuse + ISpecular;
}
```

2.3 Partikelsystem

Das Partikelsystem besteht im Wesentlichen aus zwei Klassen, einem Partikel Emitter und einem Partikel. Der Partikel Emitter wird dazu benötigt, eine gewisse Anzahl an Partikel zu verwalten. Diese Anzahl wird dem Emitter im Konstruktor übergeben. Durch eine update Funktion des Emitters werden die Partikel neu berechnet. Konkret besitzt jedes Partikel folgende Eigenschaften, die aktualisiert werden:

- Position
- Farbe
- Größe
- Aktuelle Lebensdauer
- Geschwindigkeit, mit der das Partikel wegfliegt

Wir haben die Partikel im Partikel Emitter durch eine Liste der STL dargestellt.

Neue Partikel einfügen
Um neue Partikel in den Emitter hinzufügen zu können, gibt es eine BuildParticle() Methode. Diese Methode erstellt zufällig die Position, von dem aus das Partikel startet,

die Geschwindigkeit mit der das Partikel wegfliegt, die Farbe, die Lebensdauer, die ebenfalls durch Zufall ermittelt wird, die Größe und in welchen Schritten das Partikel kleiner werden soll. Diese Methode rufen wir auf, nachdem es eine Kollision zwischen Kugel und Artefakt gegeben hat.

Rendern der Partikel
Die Partikel werden als ein Dreieck gezeichnet zwischen glBeginn(GL_TRIANGLE_STRIP); und glEnd();. Je nachdem, ob Blending aktiviert ist, wird das Partikel transparent angezeigt.

Partikel update
Prinzipiell gibt es 3 verschiedene Zustände:

- *isBorn*
  Das Partikel wird gerade geboren, hier ist der Alphawert sehr klein und wird immer größer, damit es so wirkt, als würde das Partikel langsam entstehen.

- *isDying*
  Hier wird die Größe solange verkleinert bis sie 0 ist. Das hat den Sinn, dass das Partikel nicht sofort verschwindet, sondern langsam kleiner wird.

- *Lebenszeit*
  Es wird solange überprüft, ob die momentane Lebensdauer kleiner ist, als die Gesamtlebensdauer, die das Partikel hat. Ist die Lebensdauer größer wird der isDying Zustand aktiviert.

Zum Schluss wird die aktuelle Lebenszeit neu berechnet, die Position mit der Geschwindigkeit * deltaTime sowie der Farbverlauf aktualisiert. Die Aufgabe das tote Partikel aus der Liste zu nehmen, hat der Partikel Emitter in der update Funktion. Alle Informationen über Partikelsysteme wurde unter www.codeworx.org nachgeschlagen.

3 Besonderheiten

Weiters wird in IRVANIA die komplette Szene mithilfe folgender Dateistruktur dynamisch geladen. Hierfür verwenden wir FBX-Dateien für die Geometrie und INFO-Dateien zur näheren Beschreibung.

3.1 Modellierung

Da wir am diesjährigen Maya-Kurs teilgenommen haben und die Implementierung unseres Spieles mit dem Programmieren eines FBX-Importers begonnen haben, haben wir uns dazu entschieden Maya als Modellierungs-Tool zu verwenden. Es wurden alle Objekte selber mit Maya erstellt. Auch das Titelbild wurde mit Maya erstellt.

3.2 FBX-Loader

Vor Beginn der eigentlichen Implementierung, haben wir einen einfachen FBX-Importer geschrieben, um sofort mit den richtigen geometrischen Objekten arbeiten zu können. Ein Vorteil von FBX ist, dass es ein sehr offenes Format ist, welches von sehr vielen Modellierungswerkzeugen unterstützt wird. Weiters können in der FBX-Node-Struktur auch Animationen festgehalten werden.

Sämtliche FBX-Dateien befinden sich im Ordner MODELS.

3.3 INFO-Dateiformat

Zusätzlich zu unseren FBX-Dateien speichern wir weitere Informationen zu jedem 3D-Objekt. Diese Informationen beziehen sich großteils auf Informationen über die Texturierung, Global-Position und je nach Objektart benötigte weitere Informationen. Jedes Objekt kann eine eigene Textur haben. Hierfür halten wir zu jeder FBX-Datei eine gleichnamige INFO-Datei, welche ihre Informationen im INI-Dateiformat speichert.

Die INFO-Datei befindet sich im selben Verzeichnis wie die dazugehörige FBX-Datei und hat genau denselben Namen.

Folgende Informationen können ausgelesen werden:

- *[Allgemein]*
  In diesem Punkt wird beim Startscreen die Koordinaten von Play und Exit gespeichert. Die map.info enthält unter diesem Punkt die Größe des Spielbretts.

- *[Texturierung]*
  Hier können die Texturen angegeben werden. Liegen nun mehr Objekte in einer fbx Datei, so stehen in der INFO Datei alle Texturen nacheinander immer um den Wert 1 erhöht. Da Maya die Texturen in alphabetischer Reihenfolge in die fbx Datei exportiert, besitzt Objekt1 die Texturdatei1 usw.

- *[Mauern]*
  Dieser Abschnitt beschreibt die Mauern. Zuerst gibt man an, wie viele Mauern eingelesen werden sollen und gleich dahinter muss dann unter [MauerX] die

Größe und die Position durch x, y und z Werte angegeben werden. Das X steht hier für eine laufende Zahl, die bei 1 beginnt.

- *[Artefakte]*
  Analog wie bei den Mauern gibt man zuerst die Anzahl der Artefakte an und danach müssen genauso viele [ArtefaktX] Tags dahinter liegen. Auch hier werden die x, y und z Werte eingelesen.

- *[Startpunkt]*
  Der Startpunkt wo die Kugel hinfallen soll.

- *[Endpunkt]*
  Der Endpunkt besagt die Position des Blockers und des Zylinders, der das Ziel darstellt, wenn der Blocker verschwunden ist.

- *[Hintergrund]*
  Durch diese Informationen wissen wir, wie weit die Planes für den Hintergrund vom Mittelpunkt aus entfernt sind.

3.4 Sound

Der Sound wurde mit Hilfe der FMOD Library verfasst, weitere Informationen darüber gibt es im Kapitel 4.4. Wir haben folgende Struktur erstellt:

```
typedef struct{
      FMOD::System     *system;
      FMOD::Sound      *music;
      FMOD::Channel    *channel;
      FMOD::Sound      *klirr;
}SoundTyp;
```

Durch music wird die Hintergrundmusik abgespielt und klirr erzeugt das Geräusch, wenn ein Artefakt aufgesammelt wurde. Alle Sound Dateien befinden sich im Ordner SOUND.

4 Verwendete Libraries und Zusatztools

Um das Rad nicht neu zu erfinden, verwenden wir einige Zusatztools sowie Libraries für Aufgaben, die nicht direkt mit dem Rendering der Szene zu tun haben.

4.1 PhysX

So verwenden wir auch den PhysX-SDK von AGEIA in der Version 2.7.0 für unsere Spielphysik. Hauptsächlich benutzen wir diese Library zur Collision Detection bzw. um die Kugel durch die Welt zu rollen.

Herstellerseite:
http://www.ageia.com/

Verwendete Unterlagen:
Als Informationsquelle für die Implementierung von Spielphysik mit PhysX haben wir hauptsächlich die mit AGEIA PhysX mit ausgelieferte Windows-CHM-Datei und einige Tutorials verwendet.

4.2 GLUT

Weiters verwenden wir GLUT, um unser Game-Window zu initialisieren. Genau genommen handelt es sich um die von NAME erweiterte Version VERSIONSNUMMER, da die auf der GLUT-Homepage angebotene Version leider keine Mausrad-Abfragen unterstützt.

Herstellerseite:
http://www.realmtech.net/opengl/glut.php

Verwendete Unterlagen:
Wir haben ausschließlich die dem GLUT-Framework beigelegten Hilfe-Dateien verwendet.

4.3 GLEW

Wir verwenden weiters GLEW zum Laden von einigen OpenGL-Extensions (ARB-Erweiterungen). Diese Erweiterungen verwenden wir Großteils für Vertex-Buffer-Objects und Frame-Buffer-Objects.

Herstellerseite:
http://glew.sourceforge.net/

Verwendete Unterlagen:
Für Informationsmaterial haben wir uns ausschließlich auf die CG23 Laborübungs-Homepage beschränkt, da hier alles erwähnt wird, was wir benötigt haben.

4.4 FMOD Ex

Diese Library wird verwendet um den Sound einspielen zu können.

Herstellerseite:
http://www.fmod.de

Verwendete Unterlagen:
Wir haben hauptsächlich die mitgelieferten Examples, die bei der Installation dabei sind, verwendet.

4.5 Autodesk FBX-SDK

Diesen SDK verwenden wir zum einlesen unserer FBX-Dateien in die interne Datenstruktur. Weiters verwenden wir aus diesem SDK einige Matrix- und Vektor-Klassen.

Herstellerhomepage
http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=6839916

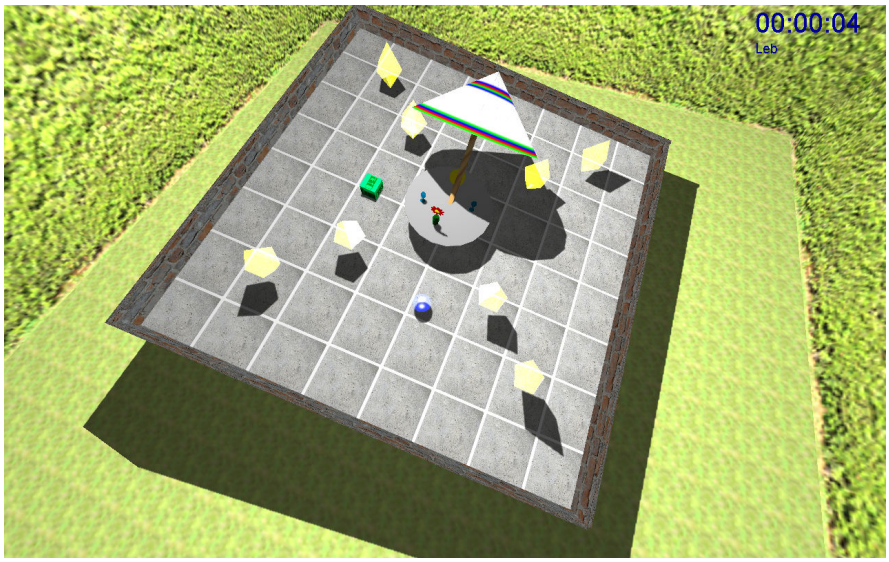
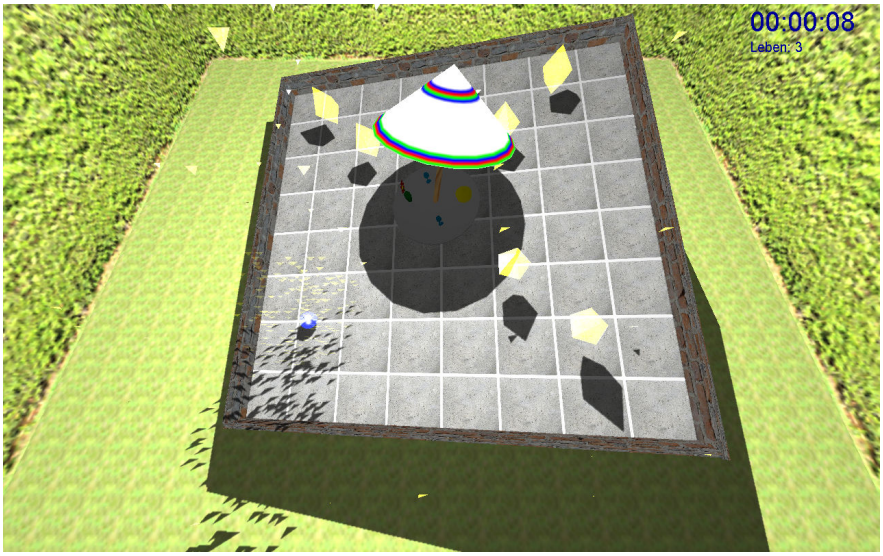
Verwendete Unterlagen
- Das auf der Herstellerhomepage befindliche Forum
- CG23LU-Repetetoriumsunterlagen

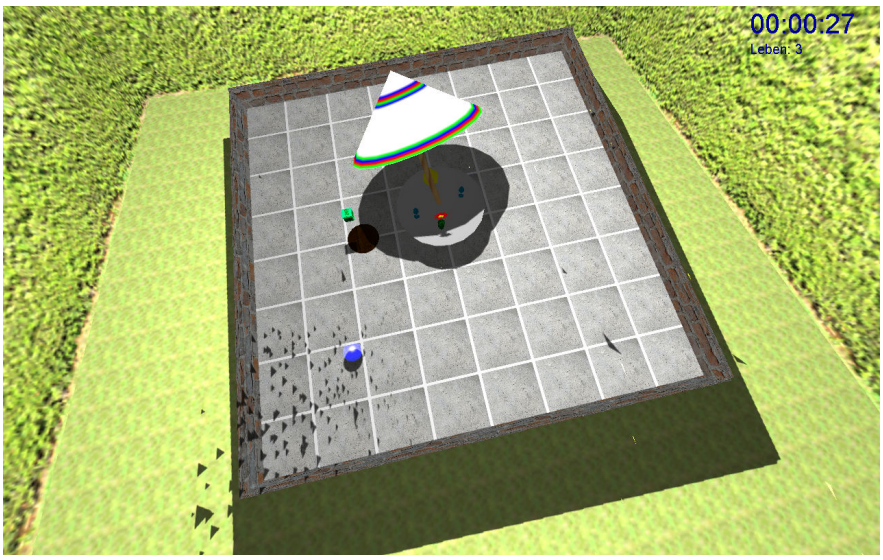
5 Walkthrough

Wir haben bis jetzt folgende zwei Level erstellt. Das Erste ist eher zum Ausprobieren gedacht, hier kann man noch nicht abstürzen. Beim zweiten Level jedoch gibt es keine Mauern mehr. Wie bereits beschrieben, müssen alle Artefakte eingesammelt werden um den Blocker verschwinden lassen zu können und das Level zu beenden.

5.1 Level 1

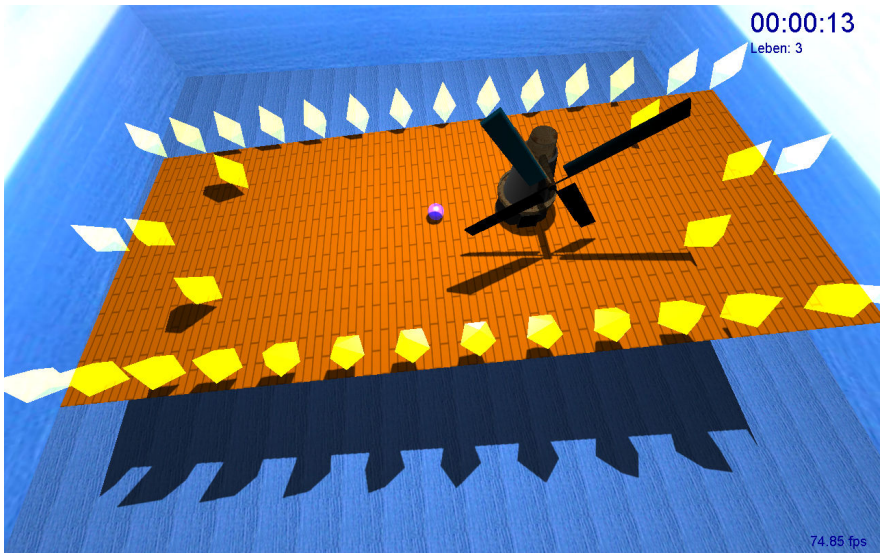
Das erste Level ist recht einfach gehalten, wie folgende Screenshots zeigen:

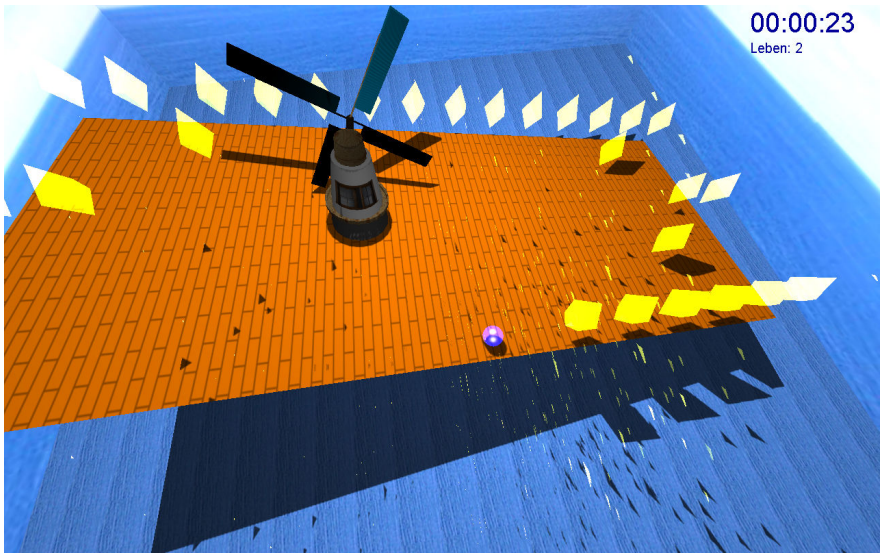
Das erste Level besitzt 8 Artefakte und der Blocker ist der grüne Würfel. Die Kugel fällt zu Beginn herunter und kann dann durch neigen des Spielbretts bewegt werden.	
Sobald Artefakte aufgesammelt werden, wird ein Sound abgespielt bzw. das Artefakt zerfällt in Partikel	

Sind alle Artefakte eingesammelt, verschwindet der grüne Blocker und das Loch wird sichtbar, durch das man ins nächste Level gelangt.	

5.2 Level 2

Das zweite Level ist etwas gefinkelter, da es keine Mauern mehr gibt und man überall runterfallen kann.

Das zweite Level besteht aus 36 Artefakten und der Blocker ist die Windmühle.	

Mitten im Geschehen…	
SIEGA! :D Alle Artefakte sind eingesammelt und die Windmühle verschwindet langsam. Das Level wurde beendet, da das Loch getroffen wurde.