

LOKI
B U R N B A B Y B U R N

3. Abgabe – LU Computergraphik 2, 186.165

Alex **DRUML** (0625181) (532)
Markus **ERNST** (0625316) (532)
Lukas **RÖSSLER** (0625652) (532)

Neuerungen in der 3. Abgabe

Frustum Culling

Objekte sowie Lichtquellen werden mit ihrer BoundingSphere gecullt.

Rendermodes/Texturemodes

Die geforderten F-Tasten Switches wurden implementiert.

Normal Mapping

Normal Mapping im per Pixel Shader implementiert. Normal Maps wurden mit CrazyBump generiert. Auf fast allen Objekten sichtbar (Terrain, Kisten, Steine).

Engine

Beliebig viele Lichtquellen (<8) können verwendet werden. Per Pixel Lighting + Volume Shadows für alle Lichtquellen.

Animation

Keyframeanimation (animierte Items). Zusehen nach der Steinwand beim Checkpoint (morphende Sphere).

Sounds

Für jede Kollision kann ein Sound spezifiziert werden, also kann jedes Paar von Objekten unterschiedlich klingen.

Schatten

Wie erwähnt für alle Lichtquellen, außerdem wurden folgende Optimierungen implementiert: Scissor Test, Depth Bounds Test, Objektculling wenn es sich außerhalb des Lichtradius befindet, Vertex Extrusion mit Infinite Projection Matrix im Vertex Shader. Die Shadow Volumes werden mit zPass gerendered nur wenn es notwendig ist (Kamera im Schatten) mit zFail.

Texturemorpher

Beim Hauptcharakter wird die Textur im Fragmentshader gemorpht um Bewegung zu simulieren.

Gameplay

- Beim Einsammeln von Energyitems facht das Feuer nun auf
- Man kann sterben (und respawnen) und natürlich auch gewinnen.
- Menüstruktur
- HUD implementiert

Multi-Threaded Loading

Szene wird in eigenem Thread geladen um bereits beim anzeigen des MainMenus mit dem Ladevorgang zu beginnen.

Geplante Erweiterungen

Sounds

Geeignete Soundeffekte für die Kollisionen zwischen den Paaren und eine unauffällige Hintergrundmusik finden.

Texturen

Detailreichere Texturierung des Terrains mittels Vertex Paint – der derzeitige Stand ist in der folgenden Grafik dargestellt:



Walkthrough

Zuerst auf den Hügel direkt vor dem Start, am besten über die Kistentreppe etwas rechts. Dann dort die Strohhaufen einsammeln und die Kiste am Ende des Hügels hinunterstoßen. Die Kiste dann an die Wand rechts von den Steinen schieben (Turbo + Anlauf verwenden) und über diese Kiste hinaufspringen. Items einsammeln und die Kiste am Ende hinunterstoßen und an die Wand rechts daneben schieben. Wieder über die Kiste hinaufspringen und auf den Checkpoint und Items einsammeln. Danach durch den Steinhaufen durch, am besten mit Turbo + springen. Grade aus zu den Steinen und wieder Items einsammeln und auf den Checkpoint. In der Ecke wo die Fackel steht ist ein Aufgang, dort hinauf und über den Bogen drüber. Turbos einsammeln und über die Rampe mit Turbo + springen das Monument umstoßen. Dann wieder über die Rampe auf das Monument und über die Klippe drüber. Danach dem Tal entlang und auf den Turm hinauf, dort die Rampe runter und über den Berg springen. Auf den Checkpoint und dann über die Säulen in Richtung Höhle springen. Durch die Höhle durch und über die Rampe am Ende mit Turbo hinaus. Dann seit ihr kurz vor dem Ziel, noch über den Abgrund mit dem Bouncer an der Wand springen und dann nur noch die Kistentreppe rauf zum Zielcheckpoint.

So ist das ganze von uns gedacht, wobei es durchaus andere Wege gibt das Level zu schaffen. Ein guter Tipp ist der „Instant-Bounce“, wenn ihr einfach Space haltet springt ihr immer direkt beim Aufprall weiter. Falls ihr mal Hilfe braucht und Turbo oder Energie aufladen wollt beim drücken von NumPad+ wird beides aufgefüllt. ;)

Falls an manchen Stellen zu Performance Problem auftreten benutzt den Shadow Switch(2).

Auf manchen System hat sich der Geometry Shader für Schatten als langsamer herausgestellt als die CPU Variante auf 6 gibt es einen Switch für diesen. Animierte Objekte werfen mit dem Geometry Shader derzeit noch keine Schatten.

Implementierungsbeschreibung

Map-Loader

Laden und Serialisieren

Wir haben uns für OgreXML als Modelformat entschieden, hauptsächlich wegen des schön lesbaren Formats und der Vielzahl an verfügbaren Exportern. Erstellt/Bearbeitet werden unsere Models in 3ds Max, exportiert anschließend mit OgreMax. Ein Loader für dotscene, .material und .mesh.xml wurde implementiert. Da das Parsen von großen XML Files sehr langsam ist und beim Laden zusätzlich einige Dinge berechnet werden müssen (Bounding-Box bzw. -Sphere usw.), werden die Meshes nach dem vollständigen Laden in unsere Datenstruktur automatisch serialisiert und als .mesh.lotm (**L**oki **T**ext **M**odel) gespeichert. Jeder weitere Ladevorgang lädt automatisch die serialisierten Files. Dieser Schritt bringt eine enorme Verbesserung der Ladezeiten (>10 Minuten zu ca. 5 Sekunden bei größeren Maps).

Erstellen der Actors (PhysX)

Damit beim Laden der Models automatisch der gewünschte Physx-Actor erstellt wird, ohne dessen Eigenschaften statisch im Source Code festzulegen, verwenden wir ein .ini File (ObjectProperties.ini) welches Profile enthält. Die Namen der Models werden in 3ds Max mit einem Suffix [profilName] versehen. Beim Laden der Models werden dann die entsprechenden Parameter ausgelesen und der gewünschte Actor erstellt. Wurde das Model mit keinem Suffix versehen wird ein Default-Actor erstellt welcher einem statischen Trianglemesh entspricht. In einem Profil kann zusätzlich angegeben werden, ob es sich um ein Item handeln soll und wenn ja, um welchen Typ von Item (Power/Turbo).

<pre>[stdBox] actorType = dynamic shape = box mass = 50.0 dimensionsX = 2 dimensionsY = 2 dimensionsZ = 2 material = wood</pre>	<pre>[woodBrick] item = power actorType = dynamic shape = convex mass = 6.0 material = wood shapeGroup = 2</pre>
---	--

Beispiele für Actor Profile

Die PhysX-Materialeigenschaften werden aus einem separaten .ini File (PhysxMaterials.ini) ausgelesen welches Profile für die Materialien enthält.

```
[wood]
restitution = 0.1
dynamicFriction = 0.2
staticFriction = 0.2
```

Beispiel für ein Material Profil

Kamerasteuerung

Ausgehend von der Anforderung den Hauptcharakter aus der 3rd Person Sicht zu steuern, wurde ein Kamerasystem mit sphärischen Koordinaten implementiert. Die Kameraposition entspricht dabei einen Punkt auf der Kugeloberfläche, der durch zwei Winkel (Azimut und Altitude) definiert ist. Durch die Bewegung der Maus werden nun die Winkel und somit die Position auf der Kugel entsprechend verändert. Der Kamerazoom (Mausrad) bewirkt eine Veränderung des Radius.

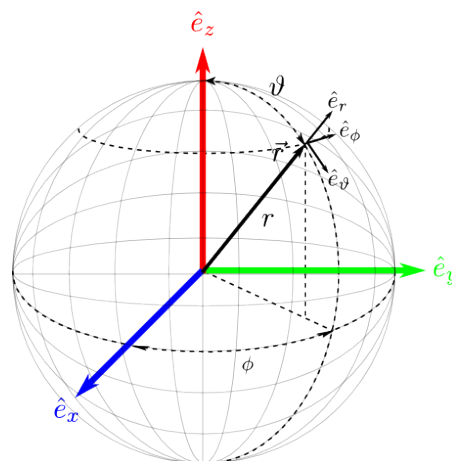


Abbildung 1 Kugelkoordinaten (1)

Die Änderung von Radius und Position bewirken eine Änderung der Zieldaten, an die sich die Kamera iterativ annähert, wodurch eine gleichmäßige Bewegung erreicht wird. Der Viewvektor (Aug-/Kameraposition \rightarrow Zentrum/Character) wird dabei nicht Linear zur Zielposition iteriert, sondern über ein Quaternion entsprechend rotiert. Als Rotationsachsen werden dafür die y-Achse und der auf die x/z Achse projizierte Normalvektor des Viewvektors und der y-Achse verwendet.

Um das „Hindurchfahren“ der Kamera durch Objekte zu verhindern wurden zwei verschiedene Methoden implementiert, einerseits durch Zoomen, andererseits durch Schwenken der Kamera. Zunächst wird ein invers zum Viewvektor und vier kegelförmig angeordnete Rays ausgesendet und mit den Objekten der Szene geschnitten. An dieser Stelle muss jedoch erwähnt werden, dass die Strahlen nur mit bestimmten Shapegroups

geschnitten werden, da es sonst z.B. bei kleinen Kisten zu einem unzumutbaren Verhalten kommt.

Sei nun x der aus Sicht des Zentrums z nächstgelegene Schnittpunkt, die aktuelle Kameraposition y und $minDistance$ der Mindestabstand der Kamera zum Objekt. Im Falle von $||z-x|| > minDistance < y$ wird der Zoomwert der Kamera auf $||z-x||$ gesetzt. Gilt jedoch $||z-x|| < minDistance$ so wird die Kamera so lange nach oben geschwenkt (rekursiver Aufruf von `UpdatePosition`) bis der beschränkende Winkel von 90° erreicht ist oder $minDistance < ||z-x|| < y$ gilt. Ergibt sich jedoch für alle Schnittpunkte $||z-x|| > y$ so wird der ursprüngliche Zoomwert wiederhergestellt.

Charaktersteuerung

Tastenbelegung



Rollen

Um den Charakter in die verschiedenen Richtungen mittels WASD zu steuern, wurde mittels der Physx-Funktion `addTorque` ein Drehmoment auf den Physx-Actor angewendet. Um dabei immer mit der W-Taste in Blickrichtung der Kamera zu rollen wird der zu addierende Kraftvektor mittels Quaternion entsprechend zur Kameravorwärtsrichtung rotiert. Die Tastaturabfrage wurde dabei über Polling realisiert wie der folgende Codeausschnitt zeigt.

```
if (glfwGetKey('W') == GLFW_PRESS)
{
    NxQuat qx(-gCamera->getForwardDirection(), NxVec3(0,1,0));

    NxVec3 forceVec =
    sceneManager->getScene(currentLevel)->getMainChar()->
```

```
getMass()*700*NxVec3(-1,0,0)*TimeManager::PHYSX_UPDATE_INTERVAL;  
  
    qx.rotate(forceVec);  
  
    sceneManager->getScene(currentLevel)->getMainChar()->addTorque(forceVec);  
}
```

Springen

Um beim Springen zu erkennen ob der Charakter den Boden „unten“ berührt wurde ein Physx-ContactReport implementiert welcher einen NxContactStreamIterator verwendet um die gemeldeten Kontakte durch zu iterieren. Dabei wird bei jedem ContactPatch die PatchNormale ausgelesen und deren Winkel zum UpVektor berechnet (0 1 0). Ist dieser Winkel zwischen 0 und 45° und handelt es sich bei dem Kontakt um einen StartTouch so wird das springen freigeschaltet, handelt es sich jedoch um einen EndTouch wird das springen gesperrt da sich der Charakter dann in der Luft befindet bis wieder ein StartTouch eintritt.

Die Sprunghöhe wird variiert je nachdem wie lange der User die Space-Taste gedrückt hält. Dies wird erreicht indem, nachdem der User begonnen hat die Space-Taste zu drücken, solange weitere Kräfte in Richtung 0 1 0 hinzugefügt werden, bis eine bestimmte maximale Kraft erreicht ist, oder der User die Space-Taste los lässt.

Bremsen

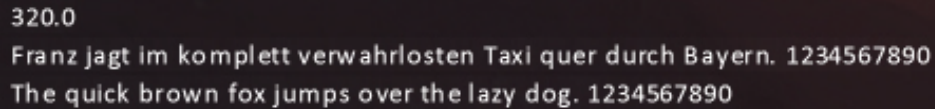
Mittels der rechten Maustaste ist es möglich die Kugel zu bremsen. Dies wird durch setzen der maximalen Winkelgeschwindigkeit auf 0 erreicht. Drückt der User zusätzlich zur rechten Maustaste noch eine Taste um eine Richtung zu rollen wird die maximale Winkelgeschwindigkeit auf einen kleinen Wert limitiert um eine sehr feine Steuerung der Kugel zu erlauben.

Turbo

Beim Turbo wird derselbe Mechanismus wie beim Springen ausgenutzt um zu detektieren ob die Kugel Kontakt hat. Ist dies der Fall, wird ein starker Kraftimpuls, sowie ein starkes Drehmoment, in Kamerablickrichtung addiert. Hat die Kugel keinen Kontakt wird nur das Drehmoment addiert.

Fonts

Um Fonts zu rendern wurde ein einfacher BitmapFontRenderer implementiert. Die Textur für diese Fonts wird mittels Bitmap Font Builder erstellt. Dieser erstellt zusätzlich ein .ini File welches die Breiten der einzelnen Buchstaben enthält. Diese Information wird dazu benutzt die Abstände zwischen den einzelnen Zeichen anzupassen. Die einzelnen Buchstaben werden als orthogonal Projizierte Quads gezeichnet deren Überlappung mittels der erwähnten Buchstabenbreite gesteuert wird. Eine Methode welche als Übergabeparameter den zu rendernden Text, sowie die Größe der Quads und die xy-Koordinaten erhält wurde implementiert.



320.0
Franz jagt im komplett verwahrlosten Taxi quer durch Bayern. 1234567890
The quick brown fox jumps over the lazy dog. 1234567890

Abbildung 2 - FontRenderer Beispiel

Items einsammeln

Damit „aufsammelbare“ Items mit anderen Objekten kollidieren können und den Charakter durch das Einsammeln nicht zu bremsen (Kollision) wird für jedes Item ein Physx-TriggerShape in der Form einer Kugel, welche einer leicht vergrößerten BoundingSphere entspricht, erzeugt. Über einen Physx-TriggerReport wird dann die Kollision des Charakters mit dem jeweiligen TriggerShape eines Items gefiltert und das Item deaktiviert (wird zu Kinematic Actor ohne Kollision). Der Charakter wird dabei über seine ShapeGroup identifiziert da dieser der einzige in Gruppe 1 ist.

Auf den Kisten rechts nach dem Start liegen ein paar Holzziegel welche man einsammeln kann.

Beleuchtung/Materialien

Alle Objekte in unserer Szene sind derzeit texturiert dabei wird aber auch jedem Objekt ein Material zugewiesen (z.B. rotes emissive Material auf dem Hauptcharakter) welches aus dem zugehörigen .material File der geladenen Szene ausgelesen wird. Die einzige Lichtquelle der Szene befindet sich an der Position des Hauptcharakters. Zusätzlich wird ambientes Licht verwendet um die Orientierung im Level zu erleichtern.

Effekte

Particelsystem

Die Simulation des Feuers erfolgt durch die Verwendung des Fluid-Emitter Prinzips von PhysX. Da hierzu mehrere Emitter benötigt werden, wurde ein Manager (*ParticleController*) implementiert, der das Laden von mehreren Emittern und Fluids unter Verwendung von Profilen ermöglicht. Ein Profil ist dabei durch einen Sectionnamen (String zwischen eckigen Klammern) in der *Loki.ini* definiert. Für eine Emitter / Fluid Kombination muss beim Erzeugen ebenfalls ein eindeutiger Name definiert werden, um diese dann später anzusprechen zu können. Das Laden von zwei unterschiedlichen Emittern zeigt das folgende Sample:

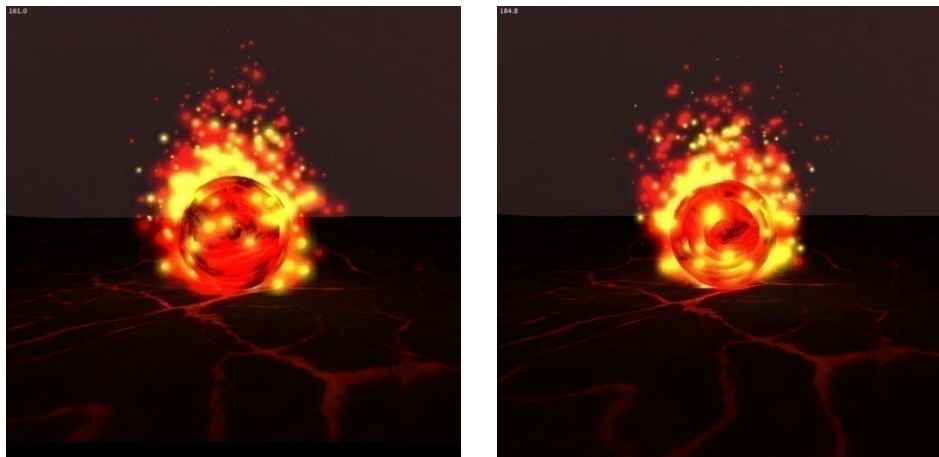
```
particleSys->addParticleEmitter("red","emitter1","fluid1");  
particleSys->addParticleEmitter("yellow","emitter2","fluid2");
```

Ein Problem stellte die Ellipsenform des Emitters dar, da es besonders bei der Bewegung der Kugel zu einer unpassenden Form der Flamme kam. Dieses Problem wurde durch Rotation der Emitterfläche beseitigt und führte zudem zu einer wesentlich besseren Form der Flamme im Stillstand und bei Bewegungen.

Zur realistischeren Darstellung des Effektes wird zudem ein Forcefield verwendet, das die Partikel zu einer Spitze formen soll. Das Laden und Erzeugen des Forcefields erfolgt ebenfalls über den ParticleController und erfordert lediglich die Zuordnung der Emittoren/Fluids auf die das Forcefield „wirken“ soll (setzen von ForceFieldMaterial).

```
particleSys->addForceField( "mainFF", "forceField" );  
particleSys->addFFtoEmitter( "mainFF", "red" );  
particleSys->addFFtoEmitter( "mainFF", "yellow" );
```

Vergleich mit bzw. ohne Forcefield:



Für das Rendern der Partikel wird additives Blending (mit Alpha) verwendet, um vor allem in Bereichen mit einer sehr hohen Partikeldichte einen helleren Farbeindruck zu erzielen.

Schatten

Die einzige Lichtquelle im Spiel soll der Hauptcharakter selbst sein. Da sich diese Lichtquelle ständig bewegt, sind Schatten ein wichtiger Effekt. Wir haben unsere Schatten mit Shadow Volumes mit z-fail Technik implementiert. Dabei wird, falls der Videotreiber OpenGL Version ≥ 2.0 unterstützt, die `glStencilOpSeparate` Funktion verwendet, weil die Volumes dann nur einmal in den Stencil Buffer gerendert werden müssen. Wird diese Funktion vom Treiber nicht unterstützt, werden die Volumes (wie herkömmlich) zweimal gerendert, einmal mit `glCullFace(GL_FRONT)` und einmal mit `glCullFace(GL_BACK)`. Nachdem in unserem Spiel nur eine Lichtquelle vorkommt kann die übliche Shadow Volumes Vorgehensweise (ambient pass -> shadow volumes in stencil buffer rendern -> light pass wo stencil == 0) noch weiter vereinfacht werden. Wir rendern die komplette Szene inklusive Beleuchtung, rendern die Shadow Volumes in den stencil buffer und zeichnen anschließend ein halbtransparentes schwarzes Quad über den ganzen Bildschirm, und zwar nur genau dort, wo stencil $\neq 0$. Dieser Hack spart uns einen kompletten Renderpass der Szene mit dem Nachteil, dass zum Beispiel Schattenüberlappungen nicht korrekt dargestellt werden können (was bei einer einzigen Lichtquelle sowieso nicht auftreten kann). Ein weiterer Nachteil ist, dass die Helligkeit der Schatten nicht exakt dem Ambient Light entspricht, sondern abhängig von der Farbe des Schattenquads ist.

Da für die z-fail Methode komplett abgeschlossene Shadow Volumes notwendig sind (inklusive Front- und Backcaps) werden die Volumes bei uns folgendermaßen berechnet: Für jedes Face des Meshs wird bestimmt ob es (von der Lichtquelle aus) ein Front- oder ein Backface ist. Frontfaces werden unverändert in den Stencil Buffer gerendert (Alle Frontfaces zusammen bilden das Frontcap). Bei Backfaces wird für jeden Vertex der Vektor von der Lichtquelle zu diesem Vertex berechnet und der Vertex in Richtung dieses Vektors um 500 Einheiten von der Lichtquelle weggeschoben. Alle Backfaces bilden gemeinsam das Backcap. Um Seitenteile des Shadow Volumes zu rendern wird nun durch alle Edges iteriert, für jede Edge bestimmt ob diese Edge ein Teil der Silhouette ist (also ob an dieser Edge genau ein Front- und ein Backface anliegen), und falls ja wird aus den beiden Vertices dieser Edge ein Quad konstruiert, indem die beiden Vertices wie vorher bei den Backfaces in Lichtrichtung um 500 verschoben werden. Diese vier Vertices (zwei an den Originalposition der Edge, zwei verschobene) bilden dann einen Seitenteil des Shadow Volumes. Beachtet werden muss beim Rendern der Shadow Caps, das die Depth Function auf GL_LESS gesetzt ist, da sonst die Frontcaps die eigentlichen Oberflächenpolygone des Objekts im Depth Test schlagen, und eigentlich beleuchtete Flächen dann schattiert würden.

Da die Berechnung der Shadow Volumes auf der cpu ausgeführt werden muss, sind die Schatten auf cpu-limitierten Systemen unser größter Bottleneck. Um diesen etwas zu entschärfen werden Objekte, die zu weit von der Lichtquelle entfernt sind, schon vor der Berechnung der Volumes gecullt, da die Schatten dieser Objekte ohnehin nicht sichtbar wären.

Einstellungen

Über das Configfile Loki.ini können einige Einstellungen vorgenommen werden unter anderem Vsync, Fullscreen und Auflösung.

Libraries

INI parser

Um *.ini Dateien zu laden wird die sehr einfache Library „iniParser“ (2) verwendet. Diese Library enthält in der aktuellen Version zwar einige Bugs, die allerdings leicht behoben werden konnten (Inkonsistenzen mit „const“ bei Parametern, einige Compiler Errors (Funktionen die im Header, aber nicht in der Implementierung umbenannt waren...)).

XML parser

Um das OgreXML Format importieren zu können war ein XML parser notwendig. Die Library „XMLparser“ (3) ist sehr einfach und erfüllt ihren Zweck.

Boost Serialization

Da das parsen der Models im xml Format sehr langsam ist, und zusätzlich auch noch verschiedene Berechnungen durchgeführt werden müssen (Berechnung von Bounding Box + Sphere, Erstellung einer Datenstruktur für die Edges, Erstellen alle Edges mit Pointern auf die

korrekten Vertices und Faces usw) wird das fertige Modellobjekt nach dem Laden vom xml mittels der Boost Serialization Library (4) serialisiert. Geladen werden können auch im fertigen Build beide Formate, .lotm (=Loki Text Model) wird aber präferiert. Vom xml wird nur geladen, wenn falls das .lotm file entweder nicht existiert oder eine falsche Serialisierungs-Version hat (sich also an der Datenstruktur etwas geändert hat).

GLFW

Window , Input und Thread Handling wird von GLFW (5) erledigt.

GLEW

OpenGL Extension Support kommt von GLEW (6).

Nvidia PhysX

Sämtliche Collision-Detection und -Response sowie die Raycasts für die Cameracollision und die gesamte Partikelphysik wird von Nvidia PhysX (7) übernommen.

DevIL

Zum Laden von Bilddateien wird die Image Library DevIL (8) verwendet.

OpenAL

Audiomischung vom 3d-Sound und Ausgabe erledigt OpenAL (9).

log4CXX

Als Logger wird der log4j Port log4cxx von Apache (10) verwendet. Auch hier musste ein bisschen im Source herumgebastelt werden bis die Library das tat was wir wollten. (DailyRollingFileAppender erstellt per Default nur dann ein neues Logfile, wenn das Programm genau zum Zeitpunkt des Rollovers (Mitternacht in unserem Fall) läuft, wie das auch bei älteren log4j Versionen der Fall war. Unsere Version der Library führt den Rollover auch durch wenn das Programm zum Rolloverzeitpunkt nicht läuft.)

Sha1

Sämtliche Konfigurationsfiles können durch SHA1 Hashes vor Modifikationen geschützt werden. Zur Berechnung der Hashes wird die Library (11) verwendet. Zum Schutz der Konfigurationsfiles existiert ein File „security“, in dem für jedes .ini File ein Hash (der durch | Verknüpfung des Files mit einem secret key erzeugt wird) angegeben ist. In der derzeitigen Version des Programmes ist der check deaktiviert, das heißt es können sämtliche Konfigurationsfiles verändert werden und das Programm schreibt den neuen Hash beim nächsten Start in das security file. Wird der Hash check aktiviert, würde das Programm bei einem veränderten Konfigurationsfile nicht mehr starten. Zusätzlich werden Hashes beim Laden der Models verwendet, um Veränderungen im xml zu erkennen und automatisch ein neues .lotm zu erzeugen.

Quellenverzeichnis

1. **Kolb, Stefan.** Wikipedia. *Kugelkoordinaten*. [Online] 08. 09 2008.
http://de.wikipedia.org/w/index.php?title=Datei:Kugelkoordinaten_svg.png&filetimestamp=20080903135133.
2. **Devillard, Nicolas.** iniParser: stand-alone ini Parser library in ANSI C. [Online] 03. 01 2008.
<http://ndevilla.free.fr/iniparser/>.
3. **Berghen, Dr. Ir. Frank Vanden.** KRANF site: research. [Online] 19. 12 2008.
<http://www.applied-mathematics.net/tools/xmlParser.html>.
4. **Ramey, Robert.** boost Serialization. [Online] 1. 11 2004.
http://www.boost.org/doc/libs/1_38_0/libs/serialization/doc/index.html.
5. **Berglund, Camilla.** GLFW - an OpenGL Framework. [Online] 01. 09 2007.
<http://glfw.sourceforge.net/>.
6. **Ikits, Milan.** The OpenGL Extension Wrangler Library. [Online] 11. 03 08.
<http://glew.sourceforge.net/>.
7. **NVIDIA.** NVIDIA PhysX. [Online] http://www.nvidia.com/object/nvidia_physx.html.
8. **Woods, Denton.** DevIL - A full featured cross-platform image library. [Online]
<http://openil.sourceforge.net/>.
9. **Creative.** OpenAL. [Online] 28. 05 2008.
<http://connect.creativelabs.com/openal/default.aspx>.
10. **Apache.** log4cxx. [Online] 31. 03 2008. <http://logging.apache.org/log4cxx/index.html>.
11. **Packetizer, Inc.** Secure Hashing Algorithm (SHA-1). [Online] 2009.
<http://www.packetizer.com/security/sha1/>.