MemARy
Projektdokumentation

Entwicklung eines Spieleprototyps auf Basis von Augmented Reality

14.02.2008

Rainer Köller, Robert Podschwadt, Stephan Trimper, Christian Weisenburger, Michael Zender

Inhalt

0 Einführung

0.1 Was ist Augmented Reality

Unter Augmented Reality versteht man die Anreicherung realer durch computergenerierte Sinneseindrücke. Je mehr Informationen man aus dem Videobild extrahiert desto mehr Möglichkeiten ergeben sich, computergenerierte Sinneseindrücke zusätzlich zu erzeugen. So können zum Beispiel in einem Videobild quadratische Marker erkannt werden und anhand ihrer Verzerrung die räumliche Lage relativ zur Kamera berechnet werden und somit diese Information dazu benutzt werden um dreidimensionale Objekte über die im Videobild erkannten Marker zu rendern.

0.2 Ziel des Projektes

Das Ziel dieses Projektes war die Evaluierung von Möglichkeiten für eine Augmented Reality Anwendung im Rahmen eines Softwaretechnikpraktikums. Des Weiteren sollte nach Möglichkeit basierend auf der Anreicherung von Videobildern durch Augmented Reality ein Prototyp für ein Spiel realisiert werden. Außerdem sollte die Applikation so gestaltet werden, dass sie als Framework von zukünftigen Projektgruppen benutzt werden kann um neue Anwendungen zu programmieren. Um die Umsetzung nicht unnötig zu verkomplizieren entschlossen wir uns als Grundlage für den Prototyp das Spiel Memory zu benutzen und dabei dem Spieler zusätzlich zu den benutzten Karten über ein Display eine dreidimensionale Darstellung der Symbole auf den Karten anzuzeigen.

1 Evaluierung

1.1 Bibliothek zur Bilderkennung

Ein Teil unserer Aufgabe in diesem Projekt war es eine geeignete Bibliothek für die Bilderkennung zu finden und zu benutzen, da der Aufwand die Bilderkennung selbst zu schreiben für den den Umfang des Projekts zu groß gewesen wäre. Auf Empfehlung unseres Betreuers Jürgen Butz haben wir uns zunächst mit der Bibliothek OpenCV beschäftigt.

1.1.1 OpenCV

OpenCV ist eine kostenlose open source Bibliothek von Intel, welche verschiedene Methoden zur Bildverarbeitung und –manipulation anbietet. Diese Möglichkeiten bietet OpenCV auch, wenn man als Bildquelle eine Kamera zur Verfügung hat, die kontinuierlich Bilder liefert.

Der erste Schritt unserer Arbeit war es die Filtermöglichkeiten von OpenCV zu nutzen um quadratische Objekte im Kamerabild zu erkennen. Dazu wendeten wir verschiedene Schwellwert- und Kantenfindungsfilter an. Mit Hilfe eines Konturfilters haben wir es anschließend geschafft Konturen mit 4 Eckpunkten aus allen gefundenen Konturen herauszufiltern. Da aber die Qualität der Filterergebnisse nicht unseren Anforderungen genügte waren wir gezwungen uns nach einer alternativen Bibliothek für die Patternfindung umzusehen.

1.1.2 ARToolkit

Bei unserer Suche nach einer Alternative für die OpenCV Bibliothek fanden wir eine ebenfalls kostenlose open source Bibliothek namens ARToolkit. Diese Bibliothek übernimmt das komplette Erkennen der Marker. Das heißt sie sucht im Bild nach schwarzen Quadraten, erkennt anhand der darin enthaltenen Symbole um welchen Marker es sich handelt und berechnet aus der Verzerrung und Skalierung des Quadrats dessen Lage im Raum relativ zur Kamera. Zur Darstellung von Objekten bietet ARToolkit zusätzlich eine Wrapperbibliothek für OpenGL Befehle um schnell an sichtbare

Ergebnisse zu kommen. Dieser Umstand war vor allem in der Test- und Einarbeitungsphase von großem Wert, da wir nicht gezwungen waren zunächst die graphische Ausgabe zu programmieren um die erzielten Ergebnisse zu verifizieren.

1.2 Kamera

1.2.1 Auswahl der geeigneten Kamera

Für die ersten Versuche mit den unterschiedlichen Bibliotheken standen zunächst etwas betagtere Webcams (Logitech QuickCam) zur Verfügung, die für bewegte Bilder (in Form eines Streams) lediglich eine maximale Auflösung von 320x240 Pixeln ermöglichten.

Schnell hat sich jedoch herausgestellt, dass Auflösung, Linsenoptik und die Qualität der CCD Sensoren für die geplante Bildverarbeitung bei weitem nicht ausreichend waren.

Aus diesem Grund war zunächst die Anschaffung einer angemessenen Kamera notwendig.

Erste Anhaltspunkte hierzu boten die Erfahrungen der Nutzer der Bilderverarbeitungs-Bibliothek „OpenCV“, die ihre Ergebnisse für Windows, MacOS und Linux auf einer Internetseite[1] zusammengetragen hatten.

Die Grundvoraussetzungen waren hier zunächst:

- hohe maximale Auflösung
- gute Kameraoptik
- hohe Bildrate für Echtzeitverarbeitung und -darstellung

Einige Anwender hatten hier gute Erfahrungen mit professionellen Kameralösungen der Marke „Prosilica“ gemacht, welche eine Anbindung über Firewire oder Gigabit Ethernet ermöglichen.

Leider waren diese Geräte erst ab mehreren hundert bis tausend Euro erhältlich, so dass diese kostspielige Anschaffung von vornherein ausschied.

Die übrigen Nutzer hatten Versuche mit qualitativ sehr unterschiedlichen Webcams aus dem Heimanwenderbereich unternommen. Die Spanne reichte hier von No-Name Produkten, die bereits ab 20 Euro erhältlich waren, bis zu relativ teuren Geräten von Creative Labs für ca. 200 Euro.

Da jedoch mehrere Anwender von sehr guten Erfahrungen mit den Kameras der Marke Philips berichteten, welche sich durch gute Treiberkompatibilität und überdurchschnittliche Qualität der Optik auszeichneten, fiel die Wahl auf das neue Modell „SPC1300NC/00“ von Philips.

Die „SPC1300NC/00“ versprach eine optische Auflösung von echten 1,3 Megapixeln (1280x1024 Pixel) im Video-Modus, einen automatischen Weißabgleich, Bildraten bis 90 FPS sowie eine hohe Lichtempfindlichkeit. Außerdem konnte davon ausgegangen werden, dass der Philips Treiber kompatibel zum vielfach eingesetzten DirectShow von Microsoft ist.

Mit ca. 80 Euro pro Stück schein diese Anschaffung ein guter Mittelweg zu sein, welche zudem (jedenfalls laut Hersteller) auch alle Grundanforderungen erfüllte.

1.2.2 Erfahrungen mit der Kamera „SPC1300NC/00“ von Philips

Der erste Eindruck des Geräts war zunächst sehr positiv, da die Kamera in Lage war sehr scharfe und klare Bilder mit 1,3 Megapixeln zu liefern. Auch der Treiber bot eine Vielzahl von Optimierungsmöglichkeiten und funktionierte und Windows XP und Vista weitgehend problemlos.

[1] http://opencvlibrary.sourceforge.net/

Leider wurde die anfängliche Freude schnell getrübt, da bei genauerem Nachprüfen die meisten der Versprechen des Herstellers gar nicht oder nicht gleichzeitig einzuhalten waren.

Folgende Probleme traten beim Einsatz der Kamera auf:

1. **Bildrate im Videomodus**
   Die versprochenen 90 FPS (frames per second) bezogen sich scheinbar lediglich nur auf die kleinstmöglichen Auflösungen der Kamera. Bei höchster optischer Auflösung war lediglich eine maximale Bildrate von 30 FPS möglich (was jedoch prinzipiell ausreichend wäre).

2. **Bildrate bei maximaler Video-Auflösung über USB 2.0-Anbindung**
   Die Bildrate kann bei dem Gerät abhängig von der gewählten Auflösung in mehreren vorgegeben Stufen gewählt werden. Für die maximale Auflösung von 1280x1024 Pixel sind z.B. Bildraten von 5, 12, 15 und 30 FPS möglich.
   Wenn jedoch eine Bildrate über 5 FPS eingestellt ist, wird das Bild deutlich unscharf und es sind schon mit bloßem Auge große JPEG-Artefakte zu erkennen.
   Die Kamera setzt also automatisch bei höheren Auflösungen ab einer bestimmten Bildrate ein Kompressionsverfahren ein. Diese Kompression fällt jedoch sehr grob aus und wird von der Kamera scheinbar ohne Optimierungsmöglichkeiten in Hardware ausgeführt.
   Beim Vergleich der maximalen Bandbreite von USB 2.0 (ca. 50-60 MByte/sec) und der benötigten Bandbreite für 1280x1024 Pixel mit 24 Bit Farbtiefe und 30 Bildern pro Sekunde (ca. 115 MByte/sec) wird auch klar, dass diese Kompression tatsächlich eine praktische Notwendigkeit ist.
   Leider wird dieser Umstand auf den ersten Blick nicht sofort klar und ist weder im Handbuch noch in der Produktbeschreibung vom Hersteller dokumentiert.
   Zusätzlich wird die Bandbreite noch von dem Audio-Datenstrom geschmälert, den das zusätzlich in der Kamera integrierte Mikrofon ständig mitliefert und auch sich auch nicht abschalten lässt.
3. **Bildrate unter verschiedenen Lichtverhältnissen**
   Sehr auffällig war, dass die Bildrate in einer hellen Umgebung sehr stabil zu sein schien, bei dunkleren Lichtverhältnissen jedoch schnell zusammenbrach.
   Durch Funktionen wie z.B. dem automatischen Weißabgleich regeln Kamera und Software die Verschlusszeit unterschiedlich, so dass ab einer bestimmten Helligkeitsgrenze die vorgewählte Bildrate nicht mehr einzuhalten ist.
4. **Treiberprobleme**
   Die unterschiedlichen Treiber haben unter den getesteten Betriebssystemen Windows XP x86, Windows XP x64 und Windows Vista x86 weitgehend zufriedenstellend funktioniert.
   Der 64-Bit-Treiber unterschied sich jedoch stark von den 32-Bit-Treibern und ermöglichte leider nicht die Nutzung der meisten angepriesenen Zusatzfunktionen, sowie nicht die Auswahl einiger Video-Auflösungen und Bildraten.
   Mit dem 32-Bit-Treiber stürzte das Betriebssystem manchmal ab (Windows Blue-Screen) wenn die Kamera zu oft ein- und ausgeschaltet wurde.

Nach ausgiebigem Testen wurde für den Normalbetrieb des Prototyen MemARy die Kamera auf eine Auflösung von 800x600 Pixeln mit einer Bildrate von 30 FPS eingestellt.

Obwohl die benötigte Bandbreite von 42 MByte/sec unter dem USB 2.0-Maximum liegt, wurde auch hier das Bild von der Kamera komprimiert übertragen. Diese Einschränkung erfolgte vermutlich

wegen der zusätzlich übertragenden Audio-Daten sowie des USB-Protokoll-Overheads, wodurch die Bandbreit für Nutzdaten zusätzlich geschmälert wird.

Um eine ausreichende Auflösung und Echtzeitfähigkeit zu gewährleisten, war dieser Kompromiss jedoch notwendig.

1.2.3 Empfehlungen bei der Suche einer besseren Kamera

Nach fast sechsmonatigen Erfahrungen mit unterschiedlichen Kameras und der eingesetzten Software können die anfänglichen Anforderungen jetzt erweitert und durch einige Empfehlungen ergänzt werden:

- hohe maximale Auflösung (mind. 1,3 Megapixel)
- gute Kameraoptik
- hohe Bildrate für Echtzeitverarbeitung (mind. 25-30 FPS)
- keine ungewünschte Bildkompression
- kein Einsatz einer Kamera mit USB 2.0 Schnittstelle (zu geringe Bandbreite, ungeeignetes Protokoll); evtl. besser geeignet:
    - Gigabit Ethernet (hier aber evtl. hoher Overhead durch zu kleine Pakete)
    - IEEE 1394a / Firewire 400 (Bandbreite theoretisch unter USB 2.0 aber besseres Protokoll)
    - IEEE 1394b / Firewire 800 (Bandbreite bis 400 MByte/s)
    - Video-Grabbing über PCI/PCIe (teuer)
- Treibersupport für relevante Betriebssysteme vorher prüfen

1.3 3D Engine

1.3.1 ARToolkit OpenGL subroutines

Wie bereits erwähnt enthält die ARToolkit Bibliothek Wrapperfunktionen für OpenGL Funktionen. Auf diese Weise lassen sich einfache Objekte wie Würfel, Kugeln oder Teekannen sehr schnell in die Applikation einbauen um die korrekte Funktionsweise der Patternerkennung zu überprüfen. Da das ARToolkit auf OpenGL optimiert ist kann die von der Patternerkennung zurückgelieferte Transformationsmatrix nach einer Konvertierung mit Hilfe der Funktion **argConvGLPara(double para[3][4], double gl_para[16])** über den Befehl **glLoadMatrixd(GLdouble* m)** direkt verwendet werden.

1.3.2 Ogre3D – Objectoriented Graphics Rendering Engine

Ogre3D ist eine objektorientierte Grafikengine. Bei der Entwicklung wurde darauf geachtet, dass das Framework so abstrakt wie möglich bleibt. Ogre3D läuft unter Windows, Linux und Mac OS X. Die unterschiedlichen Implementierungen werden hinter abstrakten Interfaces versteckt, so dass Ogre3D sich für alle Betriebssysteme möglichst gleich programmieren lässt. Es wird weiterhin die Möglichkeit geboten durch Polymorphismus unterschiedliche Implementierungen von Funktionen aufzurufen, um z.B. die Performance für Innenlevels zu erhöhen. Ogre ist in Namespaces aufgeteilt, damit es Problemlos in anderen Anwendungen verwendet werden kann.

OGRE3D vereinfacht das Laden, Darstellen und Verwalten von vorher erzeugter 3D Geometrie, im Vergleich zu den ARToolkit OpenGL Subroutines. Allerdings ist die Einarbeitungszeit höher und die Verwendung komplexer.

1.3.2.1 Überblick

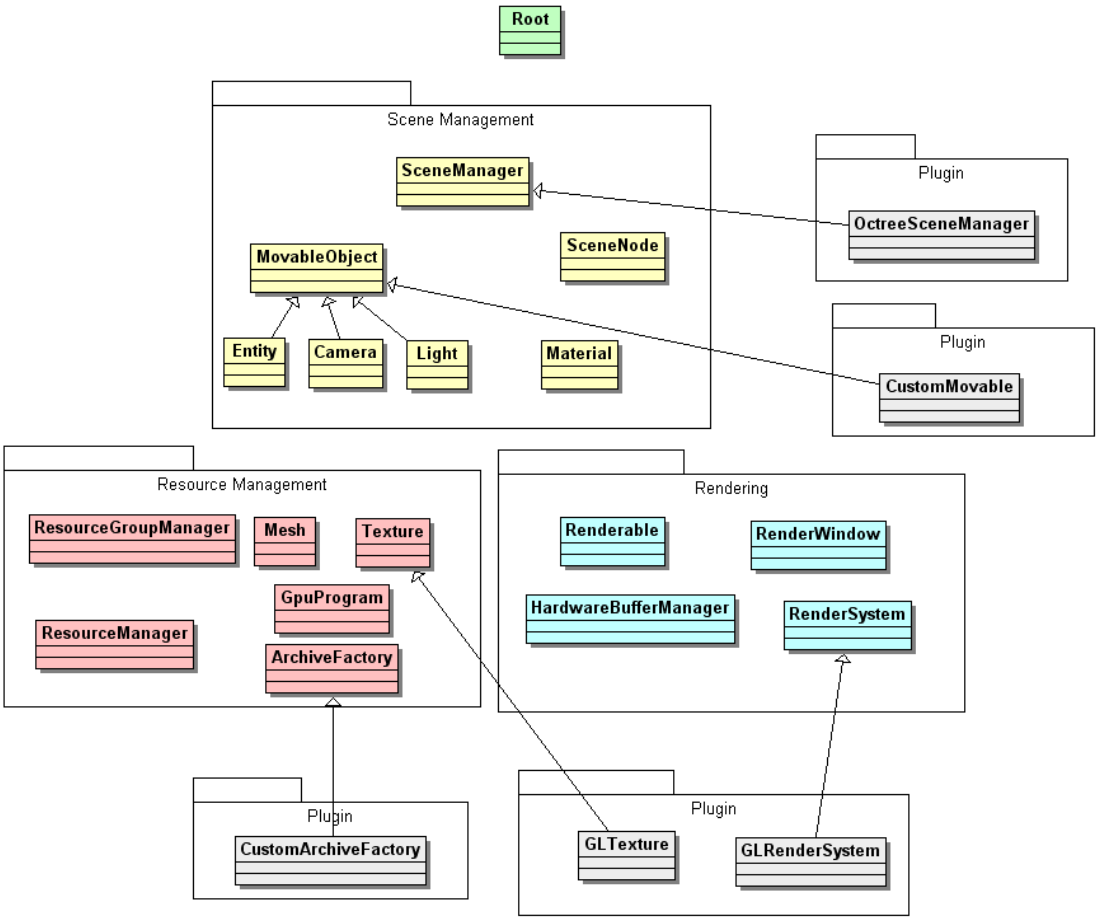


Abbildung 1 Überblick über die Klassenstruktur von Ogre3D[2]

An der Spitze des Diagramms ist das Rootobjekt. Es dient zur Verwaltung und der Erzeugung der Top-Level Objekte, wie Scenemanager, Renderingsystem, Renderwindows und für das Laden von Plugins. Der Großteil der restlichen OGRE-Klassen läßt sich in drei Gebiete einteilen:

- **Scene Management**:
  Diese Objekte sind dafür verantwortlich wie die Objekte in einer Szene angeordnet sind und wie sie von der Kamera gesehen werden.
- **Ressource Management**:
  Diese Objekte sind dafür verantwortlich, die für den Renderprozess benötigten Ressourcen wie zum Beispiel Geometrie, Texturen und ähnliches bereitzustellen und zu verwalten.
- **Rendering**:
  Diese Objekte setzen die von den Scene Management beschrieben Objekte auf ein spezifisches Rendering API um. Sie befassen sich mit den Low-Level Objekten wie Buffers und Render States.

OGRE kann in viele Richtungen erweitert werden. Normalerweise geschieht dies über Plugins (hier am Rande des Diagramms). Viele OGRE Klassen können erweitert werden um z.B. einen eigenen SceneManager zu erstellen oder ein neues RenderSystem zu implementieren (z.B. Direct3D oder OpenGL).

[2] Bild von:http://www.ogre3d.org/docs/manual/manual_4.html#SEC4

1.3.2.2 Das Root Objekt

Das Root-Objekt ist immer das Erste das erzeugt und das Letzte das zerstört werden muss. Das Root-Objekt erlaubt die Konfiguration des Systems, z.B. durch die **showConfigDialog()** Methode, die einen Dialog einblendet, in dem der User Optionen wie die Auflösung, Farbtiefe usw. einstellen kann. Das Root-Objekt bietet auch Methoden um Pointer auf andere wichtige Objekte, wie den SceneManager, das RenderSystem und andere Ressource Manager zu erhalten. Es gibt des Weiteren eine Methode, die eine durchgehende Renderingloop erzeugt, die erst abbricht, wenn alle Renderingwindows geschlossen sind oder ein FrameListener-Objekt den Abbruch der Schleife wünscht.

1.3.2.3 Das RenderSystem Objekt

Das RenderSystem Objekt ist eine abstrakte Klasse die das Interface für die darunter liegende 3D API definiert. Es gibt für jedes Renderingsystem eine spezifische Subklasse (z.B. für Direct3D oder OpenGL).

1.3.2.4 Das SceneManager Objekt

Das SceneManager Objekt ist zuständig für den Inhalt der Szene. Es organisiert und erzeugt Kameras, bewegliche Objekte (Entities), Lichter und Materials. Es verfügt dazu über Methoden wie **createCamera(), createEntity()** und **createLight()**. Der SceneManager übergibt die zu rendernde Szene an das RenderSystem wenn die Szene gerendert werden muss. Dies geschieht automatisch oder über den Aufruf von **_renderScene()**.

Es gibt unterschiedliche Subklassen, die für bestimmte Szenen die beste Performance bieten (z.B. Innenräume).

1.3.2.5 Das ResourceGroupManager Objekt

Der ResourceGroupManager verwaltet die ResourceManager für die einzelnen Resourcegruppen wie den TextureManager und den MeshManager. ResourceManager sorgen dafür, dass Ressourcen nur einmal geladen werden und dann im gesamten System zur Verfügung stehen.

1.3.2.6 Das Mesh Objekt

Ein Mesh Objekt enthält Geometriedaten, die ein Modell repräsentieren. Mesh Objekte repräsentieren meist bewegliche Objekte und werden vom MeshManager verwaltet. In einem Mesh Objekt werden die verwendeten Materialien verwaltet.

1.3.2.7 Entities

Ein Entity ist eine Instanz eines beweglichen Objekts in der Szene. Entities basieren auf diskreten Meshes die von Mesh Objekten repräsentiert werden. Mehrere Entities können auf dem selben Mesh basieren. Man erzeugt ein Entity durch den Aufruf der Methode **SceneManager::createEntity**, der der Name des Meshes übergeben wird. Der SceneManger stellt sicher, dass das Mesh geladen wird, in dem er den MeshManager aufruft. Ein Entity ist nicht dazu gedacht, Teil einer Szene zu werden bevor es nicht an einen SceneNode gehängt wird (siehe: Das SceneNode Objekt).

1.3.2.8 Materials

Material Objekte kontrollieren wie ein Objekt aussieht (abgesehen von der Form). Es spezifiziert die Oberflächeneigenschaften, wie Reflektion, Shininess, welche Texturen verwendet werden usw. Materials werden automatisch mit einem Mesh geladen.

1.3.2.9 Das SceneNode Objekt

Das SceneNode Objekt ist ein Objekt, das einen Knoten in einem Szenengraphen repräsentiert. In ihm sind Position und Transformation gespeichert. Kindknoten können eigene Transformationen haben, die relativ zu dem Elternknoten sein können, dadurch ist es sehr einfach Objekte in Abhängigkeit von einander zu positionieren. An SceneNodes können Objekte (z.B.- Lights, Entities usw.) angehängt werden und werden dadurch darstellbar. Es gibt einen Wurzelknoten, den man über die Methode **SceneManager::getRootSceneNode()** erhält. SceneNode Objekte werden durch **SceneManager::createSceneNode()** erzeugt. Das SceneManager Objekt enthält zahlreiche nützliche Funktionen, im Bezug auf SceneNodes, z.B. **getChildSceneNode().** Diese Methode liefert den Kindknoten eines SceneNodes zurück.

2 Implementation der MemARy Architektur

2.1 Anpassen der verwendeten Bibliotheken

Die für die Erkennung von Markern verwendete Bibliothek „ARToolkit“ benutzt selbst für den Zugriff auf die Kamera eine weitere Open-Source-Bibliothek mit dem Namen „DSVideoLib“[3], welche über Microsoft DirectShow von einem kompatiblen Kameratreiber Standbilder und Video-Streams anfordern kann.

DSVideoLib wiederum bedient sich einer Open-Source-Bibliothek namens „TinyXML“, welche verwendet wird, um eine kameraspezifische Konfigurationsdatei im XML-Format zu laden.

Alle drei Bibliotheken waren mehr oder weniger gut gepflegt oder wurden zum Zeitpunkt der Evaluation nicht mehr weiterentwickelt, so dass es notwendig war, selbst einige Anpassungen vorzunehmen, damit die Möglichkeit zur Nutzung mit aktueller Hardware mit der nötigen Stabilität gewährleistet war.

Im Folgenden sind alle Anpassungen kurz dokumentiert.

2.1.1 Portierung des gesamten Quellcodes der Bibliotheken in eine VS2005 Solution

Bisher lagen lediglich Projekte im Format von Microsoft Visual Studio 2003 oder früheren Versionen davon vor. Zur vereinfachten Fehlersuche, besseren Übersichtlichkeit und besseren Kompatibilität war eine Vereinheitlichung im Visual Studio 2005-Format sinnvoll.

1. Grundlage: VS2003 Solution von ARToolKit, offizielle Version 2.72.1[4].
2. Konvertierung in VS2005 Solution
   ARToolKit ist in klassischem C programmiert. Einige Headerdateien deklarieren hierin Variablen ohne Datentypen dafür zu definieren. Dies ist jedoch in C++ nicht mehr erlaubt und VS2005 akzeptiert diesen Modus nicht mehr. Es wurden nachträglich die von VS2005 vorgeschlagenen Datentypen zu den Deklarationen hinzugefügt.
3. Anpassung der Include Pfade/Library Directories an die lokalen Installationen
4. Import des DSVL Projekts (bereits in Unterordner des ARToolKit-Pakets enthalten)
5. Import des TinyXML Projekts (bereits in Unterordner des ARToolKit-Pakets enthalten)
6. Installation des Microsoft DirectX 9 Software Developer Kits (DX9SDK) (August 2007)
   Das für DSVL benötigte DirectShow ist hierin nicht mehr enthalten.

[3] http://sourceforge.net/projects/dsvideolib/
[4] http://www.hitl.washington.edu/artoolkit/download/

7. Deshalb: Hinzufügen der Microsoft DX9 Extras (Februar 2005)
   Diese enthalten die DirectShow-Beispielbibliotheken, welche von DSVL verwendet und erweitert werden. Der Ordner „Extras“, der „DirectShow“ und „DirectSound“ enthalten sollte, muss in den Stammordner der DX9SDK Installation kopiert werden.
8. Anpassung der Include Pfade/Library Directories für die DX9 Extras und DX9SDK
9. Ausschalten der „Precompiled Headers“ für alle relevanten Projekte
10. Änderung des Debug-Modus für alle relevanten Projekte auf „Multi-threaded DLL”
    Für einheitliches und fehlerfreies Kompilieren notwendig!
11. Alle Projekte umstellen auf „Compile as C++ Code“ um die Kompatibilität zwischen allen Projekten sicherzustellen
12. Änderung von „Treat wchar_t as Built-in Type“ auf „Yes“ für Projekte, die beim Kompilieren ein Fehlermeldung diesbezüglich liefern
13. Anpassung der Bibliothekspfade/Ausgabepfade damit der Debugger zwischen den unterschiedlichen Projekten hin- und herspringen kann

Ein fertig angepasstes Projekt mit allen Quelldateien sowie die nötigen Installationspakete wurden mit dieser Dokumentation zusammen zur Verfügung gestellt!

2.1.2 Bugfixing „DSVideoLib“

DSVideoLib bringt bereits eine eigene strukturierte Konfigurationsdatei im XML-Format mit, in der z.B. folgende Voreinstellungen möglich sind:

- Ein- und Ausschalten eines Konfigurationsdialogs, der vor jedem Start des Programms individuelle Kameraeinstellungen zulässt. Dies ist eine allgemeine DirectShow Funktionalität, welche von jedem kompatiblen Kameratreiber vom Hersteller implementiert werden muss. Dieser Dialog sieht bei jedem Treiber/Hersteller anders aus und bietet unter Umständen auch unterschiedliche Konfigurationsmöglichkeiten.
- Vorauswahl der zu verwendenden Video-Auflösung
- Vorauswahl der zu verwendenden Bildrate (FPS)
- Farbtiefe/Farbformat/Pixelformat
- Seitenorientierung des Kamerabilds (z.B. auf dem Kopf, etc.)

Leider ist auch in der neusten Version von ARToolKit noch eine nicht mehr ganz zeitgemäße Fassung von DSVL enthalten, da dieses inzwischen nicht mehr weiterentwickelt wird. Diese Version unterstützt lediglich einige wenige Auflösungen von frühen DV-Kameras.

Moderne Kameras bieten jedoch höhere Auflösungen und auch noch diverse zusätzliche Zwischenstufen. Bei manueller Auswahl hatte DSVL diese als fehlerhaft eingestuft und den Treiber angewiesen, die kleinste verfügbare Auflösung zu verwenden. Gleiches galt für die Anpassung der Bildrate.

Damit das volle Potential der Kamera mit ARToolKit nutzbar war, musste die Funktion „MatchMediaTypes“ in der Klasse „DSVL_GraphManager“ überarbeitet werden (für Details siehe Kommentare im Quellcode).

2.1.3 Bugfixing „ARToolKit“

Bei der Arbeit mit ARToolKit traten unter bestimmten Umständen immer wieder Fehlermuster auf, die auf diverse kleinere Programmfehler wie z.B. falsche Schleifen-Iteratoren zurückzuführen waren.

Ein großes Problem stellte jedoch die Tatsache dar, dass ARToolKit bei Videoauflösungen über 1024x768 Pixeln grundsätzlich abstürzte. Nach ausgiebiger Suche konnte festgestellt werden, dass in den Algorithmen der Bilderkennung immer von einer maximalen Bildgröße mit 1024x1024 Bildpunkten ausgegangen wurde. Nach Anpassung der entsprechenden Stellen in der Quelldatei „AR/arLabeling.c“ (siehe Quellcode) konnte aber auch dieses Problem gelöst werden.

2.2 Patternverwaltung mit XML

Zur Laufzeit benötigt die MemARy Applikation verschiedene Daten über die zu erkennenden Patterns und die ihnen zugeordneten 3D Modelle. Um eine zentrale Verwaltung dieser Daten und einen einfachen Austausch der Patterns bzw. 3D Modelle zu ermöglichen haben wir uns dazu entschlossen die benötigten Daten in einer XML-Datei zu verwalten, die während dem Initialisierungsvorgang der Anwendung eingelesen und verarbeitet wird. Die XML-Datei „patterns.xml“ muss im Verzeichnis „.\res“ liegen und für jedes Pattern folgende Angaben enthalten:

- **objectID (String)**
  Dies ist eine eindeutige String-ID, die dazu dient das Pattern innerhalb des Programms zu identifizieren. Zusätzlich wird diese Angabe dazu benutzt die Namen der in Ogre3D erzeugten SceneNodes und Entities zu generieren.
- **fileName (String)**
  Der Dateipfad des Patterns relativ zum Ausführungsverzeichnis.
- **model (String)**
  Dies ist der Dateiname des diesem Pattern zugeordneten 3D Modells. Die Angabe erfolgt ohne Pfad.
- **length (float)**
  Die Kantenlänge des Patterns in Millimetern.
- **center**
  Dieses Element hat zwei Unterelemente (x und y) wodurch der Mittelpunkt des Patterns im Intervall [0,1] angegeben wird.

Genauer muss die XML-Datei folgender DTD genügen:

```
<!ELEMENT patterns (pattern*)>
<!ELEMENT pattern (objectId, fileName, model, length, center)>
<!ELEMENT objectId (#PCDATA)>
<!ELEMENT fileName (#PCDATA)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT length (#PCDATA)>
<!ELEMENT center (x, y)>
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
```

Eine patterns.xml-Datei mit einem Pattern hat beispielsweise folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Wurzelelement SYSTEM "patterns.dtd">

<patterns>
    <pattern>
        <objectId>Buch_B</objectId>
        <fileName>..\\..\\res\\pattern\\buch_b.patt</fileName>
        <model>Buch.mesh</model>
        <length>80.0</length>
        <center>
            <x>0.5</x>
            <y>0.6</y>
        </center>
    </pattern>
</patterns>
```

Das Einlesen der XML-Datei übernimmt in der MemARy Applikation die Klasse **PatternParser**. Sie enthält die Methode **parse(char* xmlFile, list<PatternInfo*>* patternInfoList)**, welche während der Initialisierung aufgerufen wird. Diese Methode liest die Datei **xmlFile** ein und benutzt den Xerces-DOM-Parser[5] um die einzelnen Informationen aus der eingelesenen Datei zu extrahieren und diese in eine Liste aus **PatternInfo** Objekten einzufügen. Diese Liste wird zur Laufzeit bei der Initialisierung bzw. bei der Durchführung der Bilderkennung benötigt und auch verändert.

2.3 Algorithmus zur Patternerkennung

2.3.1 Erklärung des Algorithmus

Der zentrale Kern von ARToolKit ist die Wiedererkennung von bereits vorher eingelesenen „Markern“ (siehe oben). Der Software sind Größe (in Millimeter) und Form (quadratisch) der zu findenden Marker bekannt.

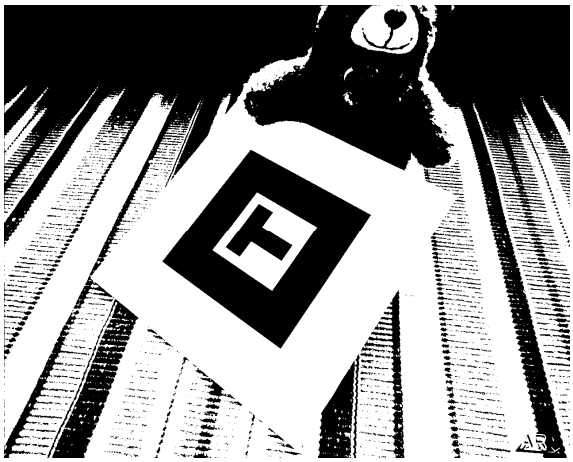
Die Bildverarbeitung wird durch Aufruf der Funktion „arDetectMarkers“ ausgeführt. Der Funktion muss unter Anderem ein Array übergeben werden, welches das zu analysierende 24-bit-Farbbild enthält. Woher das Bild kommt bleibt prinzipiell dem Anwender überlassen, jedoch enthält ARToolKit direkt eine Wrapperfunktion „arVideoGetImage“, die mit Hilfe der enthaltenden Bibliothek „DSVideoLib“ mittels Microsoft DirectShow beim Kameratreiber ein Bild anfordern kann.

Mit diesen Informationen und Hilfsmitteln läuft der Erkennungsvorgang für jedes ankommende Bild auf dem Video-Stream im Groben folgendermaßen ab:

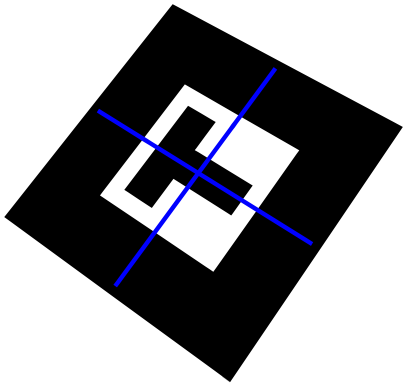
[5] http://xerces.apache.org/xerces-c

1. Einzelbild aus dem Video-Stream entnehmen

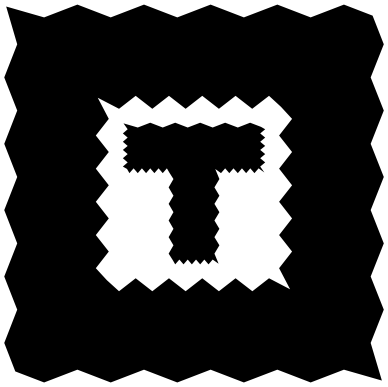
2. 1-bit SW-Konvertierung (mit variablem Schwellwert) und Bildkorrektur unter Berücksichtigung der Kameraeigenschaften (Linse, etc.)



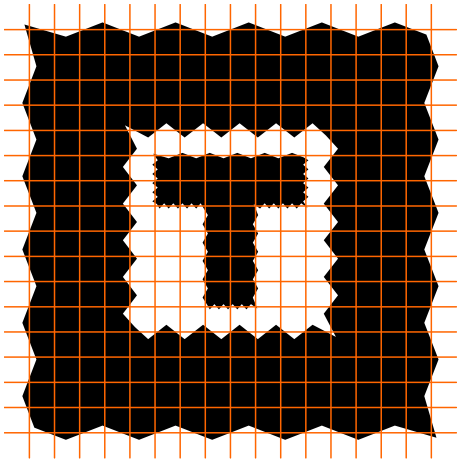
3. Schwarze Flächen/Teilflächen im Bild suchen (mit bekannten Algorithmen)



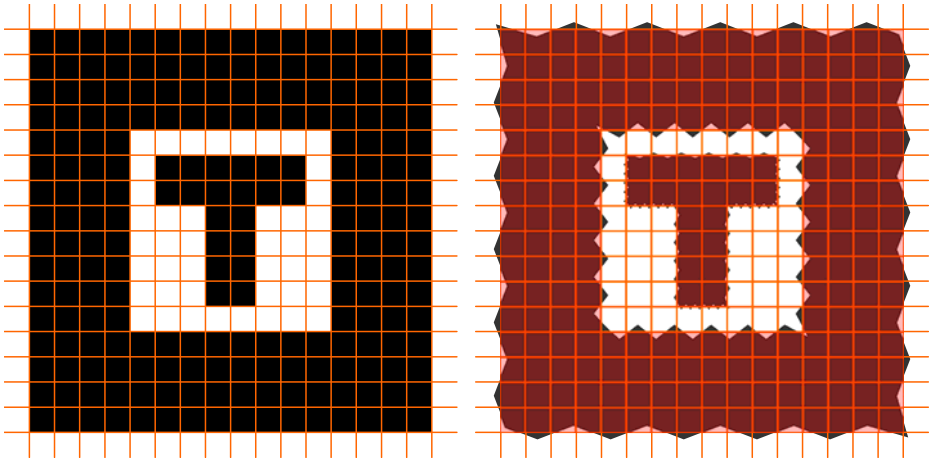
4. Diese potentiellen schwarzen Bildstellen vergrößern und so verzerren, dass ein Quadrat der gesuchten Größe damit ausgefüllt wird.



5. Den verzerrten Bildausschnitt rastern



6. Vergleich der Rasterzellen auf Bit-Ebene mit allen bekannten Mustern, dadurch Bestimmung eines individuellen Wahrscheinlichkeitswerts (hier z.B. ca. 90%).



7. Der Ausschnitt wird als erkannt markiert und als ein bekanntes Muster klassifiziert, wenn bestimmte Grenzwerte eingehalten sind.

Die vorher angewendete Transformation (Verzerrung) gibt Aufschluss über die Lage und Größe des Markers im Bild; die 3D Position kann berechnet werden

Abbildung 2 Eingeblendetes 3D Modell in Kombination mit Live Kamerabild

2.3.2 Möglichkeiten zur Ermittlung der Transformationsmatrix

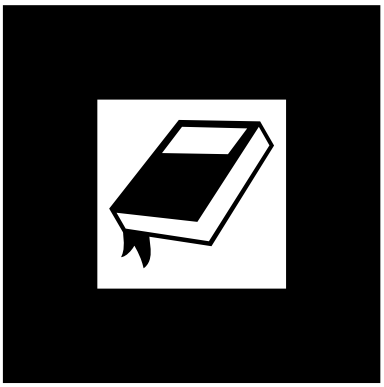
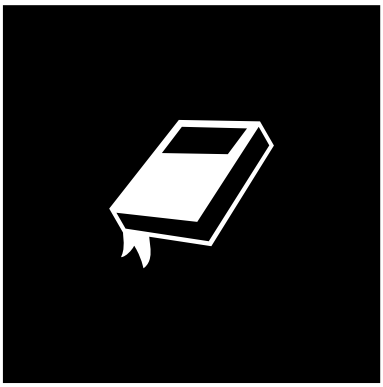
Jedes bekannte Muster erhält beim Start von ARToolKit eine eindeutige ID. Zusätzlich wird jedem erkannten Marker eine eindeutige Referenz zugeordnet. Anhand dieser Referenz kann über die Funktion „**arGetTransMat**“ die aktuelle Transformation des erkannten Markers abgefragt werden.

Selbstverständlich ist jeder Erkennungsvorgang aufgrund von Rundungsfehlern, Rauschen oder Hardwareeinschränkungen mit einem geringen Fehler behaftet. Um dies zu kompensieren und z.B. das „Wackeln“ der 3D Darstellung durch die von Bild zu Bild minimal variierenden Bildkoordinaten zu vermeiden, bietet ARToolKit die Funktion „**arGetTransMatCont**“ an. Diese Funktion stabilisiert die Koordinatentransformation unter der Annahme, dass jedes Muster nur einmal im Bild vorkommen kann. Das heißt, dass jedes Muster zu jeder Zeit nur höchstens einem Marker zugeordnet werden kann.

Die jeweils zuletzt bekannte Position desjenigen erkannten Markers, der zuletzt als ein bestimmtes Muster identifiziert wurde, wird abgespeichert und bei Verwendung dieser sogenannten „History-Funktion“ mit einer gesondert bestimmten Gewichtung einbezogen. Die Gewichtung hängt ab von der Abweichung der alten Position zur neuen Position. Ist die Abweichung zu groß, fällt der alte Wert gegebenenfalls gar nicht mehr ins Gewicht, weil sonst ein signifikanter Fehler bei der Positionierung sehr wahrscheinlich wäre. Auf diese Weise ist es möglich, weiche Übergänge bei der Bewegung von Objekten zu erreichen und sie im Stillstand sichtbar zu stabilisieren.

Dies funktioniert allerdings wie bereits erwähnt nur dann, wenn es jeden Marker nur genau einmal geben darf. Wird derselbe Marker-Typ mehrmals ins Bild gebracht, liefert diese Funktion keine korrekten Ergebnisse mehr.

Zur Realisierung des Spielprinzips von „Memory“ in diesem Prototypen MemARy konnten deshalb die sonst üblichen identischen Spielkartenpaare nicht verwendet werden. Daher wurden jeweils zwei Karten erstellt, die zwar dasselbe Symbol zeigen, von denen jedoch eine Karte das Symbol in invertierter Form darstellt. Für den Spieler ist die Bedeutung der Karten somit identisch, die Software kann die Karten aber eindeutig unterscheiden.

Unter bestimmten Umständen versagten die Algorithmen zur Positionsbestimmung allerdings leider und brachen die Berechnung mit einem negativen Rückgabewert (=Fehlermeldung) ab. Dies ist höchstwahrscheinlich auf Rundungsfehler und Ungenauigkeitsfaktoren der verwendeten Datentypen zurückzuführen.

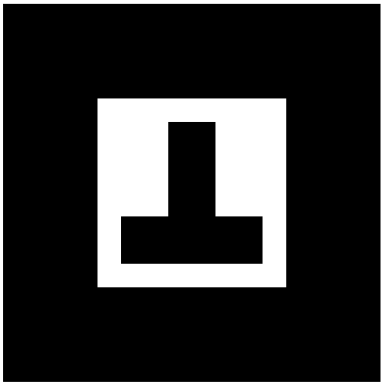
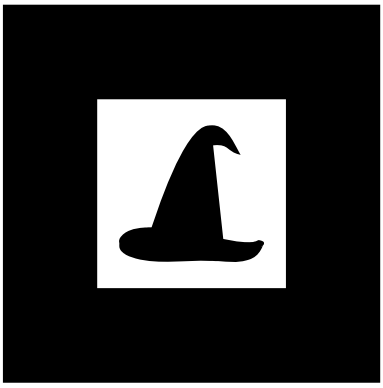
Im Falle eines solchen Fehlers wurde sich einfach mit der direkten Wiederherstellung der zuletzt verwendeten Transformation beholfen, indem vor jedem Berechnungsvorgang eine „Sicherheitskopie“ der Matrix gemacht wird.

2.3.3 Hinweise zur Erstellung von Mustern (Patterns)

Die Rasterung der Bildausschnitte ist zwar variabel einstellbar, aber im Quellcode fest eingetragen (erfordert also erneutes Kompilieren des Projekts). Die Standardeinstellung der Rasterung beträgt 16x16 Pixel und wurde auch für dieses Projekt eingehalten. Die Qualität der Erkennung (vor allem für entfernte Marker) kann erheblich gesteigert werden, wenn das Raster verfeinert wird. Jedoch ist auch zu beachten, dass die Rechenzeit schnell explodieren kann, wenn viele Mustervergleiche nötig sind.

Die Pattern-Vorlagen sollten entsprechend der Rasterung keine Ähnlichkeiten aufweisen (auch nicht unter extremen Aufsichtswinkeln oder nach Drehung).

Bei zu geringer Rasterauflösung sind diese beiden Muster z.B. sehr schnell verwechselt:

2.4 WebcamARTPlugin

Ein zentrales Problem war es das Kamerabild im Hintergrund des Ogre3D Viewports bildschirmfüllend darzustellen. Bei unseren Recherchen zu diesem Problem stießen wir auf eine Schnittstelle in Ogre3D, die es ermöglicht über ein Plugin externe Texturressourcen wie zum Beispiel Filme, Kamerabilder oder sogar Ausgaben anderer Programme in einem normalen Material als Texturquelle zu verwenden. Dieses Plugin muss als DLL-Datei erstellt werden und wird dann beim

Start von Ogre3D automatisch mitgeladen. Als Basis für ein solches Plugin steht die Klasse **ExternalTextureSource** zur Verfügung. Die Klasse **ExternalTextureSource** schreibt einige Methoden vor, welche bei der Initialisierung aufgerufen werden um die Textur zu erzeugen. Da beim Initialisierungsvorgang die Bilderkennung noch nicht gestartet ist müssen wir uns hier selbst ein Kamerabild holen um die Texturgröße genau festlegen und entsprechend Speicher reservieren zu können. In der Methode **updateTexture(unsigned char* imageData)** die im Framelistener aufgerufen wird sobald ein neues Bild von der Kamera zur Verfügung steht wird das neue Bild in den Texturspeicher kopiert um die Anzeige zu aktualisieren. Diese Textur wird auf eine in der **RENDER_QUEUE_BACKGROUND** gerenderte Plane gemapped. Da eine Textur immer quadratisch und ihre Kantenlänge eine Zweierpotenz sein muss, das Bild von der Kamera aber im Format 4:3 und je nach Einstellung und verwendeter Kamera in unterschiedlichen Auflösungen vorliegen kann, müssen die Texturkoordinaten der Hintergrundplane entsprechend angepasst werden. Um die Texturkoordinaten beliebig manipulieren zu können mussten wir zusätzlich eine von **Rectangle2D** erbende Klasse erzeugen, die diese Funktionalität zur Verfügung stellt.

2.5 SceneNodeManager

Ogre3D verwendet für die interne Repräsentation einen sogenannten Szenengraphen. Dieses Verfahren ist im professionellen 3D-Bereich sehr beliebt und ermöglicht die Verwaltung von 3D-Objekten in einer geordneten Baumstruktur.

Die Aufgabe des SceneNodeManagers sollte es sein, alle benötigten 3D-Modelle für die Darstellung vorzuhalten und deren Sichtbarkeit im Zusammenhang mit erkannten (und gerade nicht erkannten) Patterns zu steuern.

2.5.1 Initialisierung

Beim Erzeugen des SceneNodeManagers wird eine Liste aller bekannten Muster übergeben und zunächst der Abschnitt des Szenengraphen erzeugt, welcher die 3D-Modelle enthält (siehe in Grafik: „scenenode_manager“).

Für jedes Muster wird an diesem Knoten ein weiterer Unterknoten angelegt, in dem eine sogenannte „Entity“ eingehängt wird, welche das 3D-Modell enthält. Diese „Entity“ wurde aus dem zugehörigen „Mesh“ erzeugt, welches vorher vom PatternParser geladen und dem Muster zugeordnet wurde. Zusätzlich wird neben dem 3D-Modell noch eine weitere Entity eingehängt, welche eine zweidimensionale Abdeckebene zur Darstellung unter dem Modell enthält.

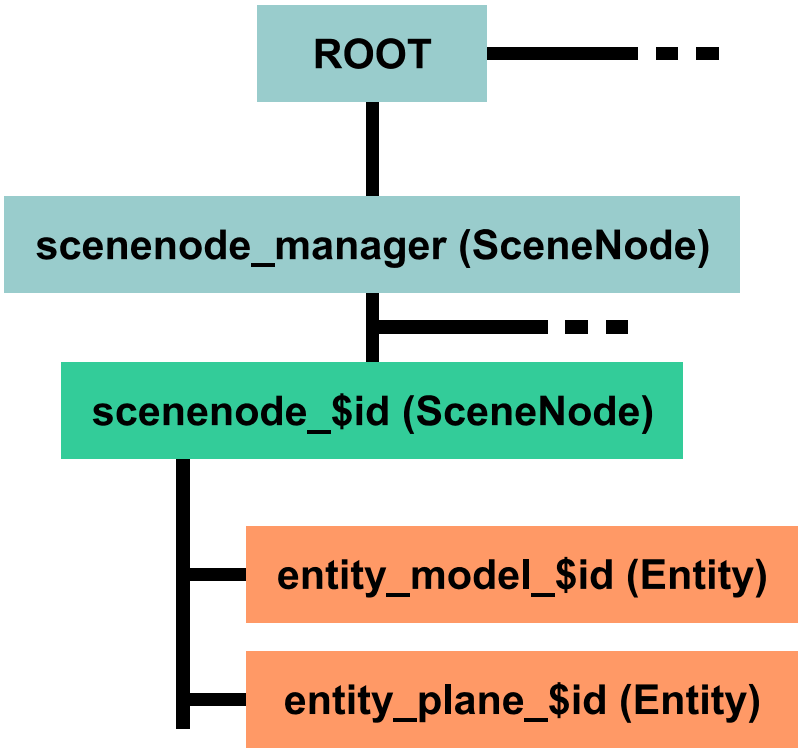


Abbildung 3 Struktur des Szenengraphen in Ogre3D mit SceneNodeManager

2.5.2 Szenengraph für die Darstellung der benötigten Objekte anwenden

Jede Entity kann über die Funktion „**setVisible**“ sichtbar oder unsichtbar gemacht werden. Auf diese Weise ist es möglich, alle benötigten 3D-Objekte nach einmaligem Erzeugen im Speicher vorzuhalten. Somit wird ständiges Löschen und Neuerstellen der Objekte vermieden, was viel Rechenzeit kosten würde.

Die Sichtbarkeitsfunktion kann zudem in rekursiver Form auf jedem übergeordneten Knoten aufgerufen werden.

Nach der Verarbeitung eines jeden Kameraeinzelbilds wird nun die Funktion „**updateSceneNodes**“ aufgerufen. Diese Funktion führt einen Abgleich zwischen der Liste der in diesem Durchgang (über die Marker) erkannten Patterns und ihren zugehörigen Repräsentationen im Szenengraph durch.

Wenn ein Pattern gefunden wurde, wird auf der zugehörigen SceneNode „scenenode_$id“ „**setVisible(true)**“ aufgerufen, wenn nicht wird „**setVisible(false)**“ aufgerufen.

Wenn das Objekt sichtbar sein soll, wird zusätzlich noch die erforderliche Umrechnung der Transformationsmatrix aus dem GL-Format, welches ARToolKit liefert, in das Format von Ogre3D durchgeführt.

Dieser Implementierung wird die Tatsache zu Nutze gemacht, dass jedes Pattern nur maximal einmal auftreten kann; Mehrdeutigkeiten gibt es nicht. Auf diese Weise kann für jedes Pattern genau eine Entity für das 3D-Objekt erzeugt werden und dann einfach ein- und ausgeschaltet werden.

2.6 Umwandlung der Transformationsmatrix von ARToolkit nach Ogre3D

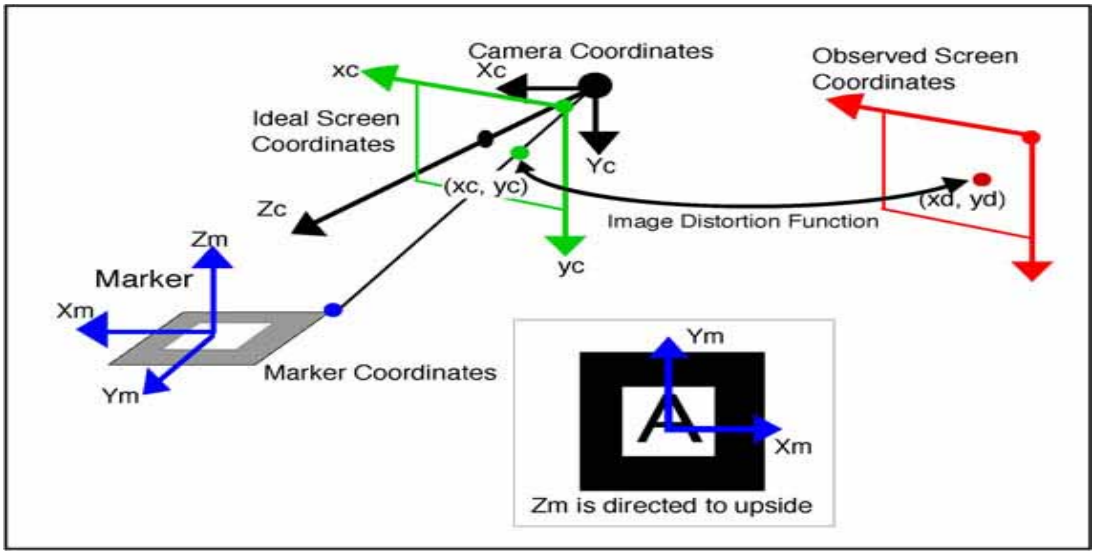


Abbildung 4 Umrechnung von Bildschirm zu 3D Koordinaten[6]

Das obige Bild zeigt die Koordinatensysteme, die ARToolkit verwendet. Die Methode **arGetTransMat()** liefert die Position des Markers im Kamera Koordinatensystem. Um nun von einem Punkt in dem Koordinatensystem des Markers auf den entsprechenden Punkt in Koordinatensystem der Kamera zu kommen muss man den Punkt mit

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & cos270° & -sin270° \\ 0 & sin270° & cos270° \end{bmatrix}$$

multiplizieren, was einer Rotation um 90° im Uhrzeigersinn um die X-Achse entspricht.

OGRE3D verwendet das rechtshändige Koordinatensystem.

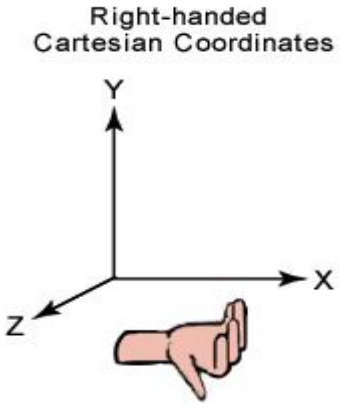


Abbildung 5 Rechtshändiges kartesisches Koordinatensystem[7]

Um diesen Punkt jetzt in dieses Koordinatensystem zu transformieren, muss man den Punkt an der XZ Ebene und an der YZ Ebene spiegeln, also mit folgender Matrix multiplizieren:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

[6] Bild von : http://www.hitl.washington.edu/artoolkit/documentation/cs.htm
[7] Bild von: http://msdn.microsoft.com/archive/en-us/directx9_m/directx/art/leftrght.gif

Daraus ergibt sich für einen Punkt auf dem Marker folgende Transformation um ihn in das Koordinatensystem von OGRE zu transformieren:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -cos270° & -sin270° \\ 0 & -sin270° & cos270° \end{bmatrix}$$

2.7 Die MemARy Architektur im Überblick

Um einen leichteren Einstieg in die Arbeitsweise der Applikation vor allem für zukünftige Projektgruppen zu ermöglichen wird diese nun kurz umrissen:

- **MemARyStarter**
  In dieser Klasse befindet sich die main-Methode der MemARy Applikation. Hier wird eine Instanz der MemARyApp Klasse erzeugt und deren **go()** Methode aufgerufen.
- **MemARyApp**
  Diese Klasse erbt von der Klasse ExampleApplication, die mit Ogre mitgeliefert wird. Diese Klasse realisiert alle zum Start einer Ogre3D Applikation nötigen Schritte. Insbesondere enthält sie die **go()** Methode, die in der MemARyStarter Klasse aufgerufen wird. In der go() Methode werden einige Methoden aufgerufen (zum Beispiel **configure()**, **createCamera()**, **createScene()** und **createFrameListener()**), die wichtige Schritte zur Initialisierung der Applikation übernehmen. Diese Methoden werden ebenfalls in der MemARyApp Klasse überschrieben und entsprechend der MemARy Applikation implementiert. Zusätzlich zu den überschriebenen Methoden enthält die MemARy Starter Klasse eine Init Methode, die zusätzliche Initialisierungen enthält. Sie initialisiert den **PatternParser** und die **ImageRecognitionARToolkit** (Bilderkennungskomponente).
- **PatternParser**
  Der PatternParser enthält die Methode **parse()**, in der die unter 2.2 Patternverwaltung mit XML genannte XML-Datei eingelesen wird. Die geparsten Daten werden von der Methode als **PatternInfo** Objekte in die Liste **patternInfoList** eingefügt, welche der **MemARyApp** Klasse gehört.
- **PatternInfo**
  Die Klasse **PatternInfo** enthält alle Daten aus der patterns XML-Datei. Zusätzlich ist hier ein Attribut vorhanden in welchem gespeichert wird ob dieses Pattern gerade sichtbar ist oder nicht (**bool isVisible**)sowie ein weiteres Attribut in welchem die interne PatternID gespeichert ist, die beim Einlesen der Pattern vom ARToolkit vergeben wird (**int id**). Außerdem gibt es noch ein Attribut, in dem die Transformationsmatrix abgelegt wird (**double transformationMatrix[3][4]**).
- **MemARyFrameListener**
  Diese Klasse erweitert die Klasse **FrameListener** von Ogre3D und wird in der **createFrameListener()** Methode in der **MemARyApp** Klasse als FrameListener registriert. Dadurch wird vor dem Rendern jedes Frames die Methode **frameStarted(const FrameEvent& evt)** aufgerufen. In dieser Methode wird die Methode zur Bilderkennung aufgerufen (**detectMarkers(…)**) und ggf. die Textur mit dem Hintergrundbild aktualisiert. Außerdem werden hier über die Methode processUnbufferedKeyInput(const FrameEvent& evt) Tastatureingaben verarbeitet und der Szenengraph aktualisiert **(mSceneNodeManager->updateSceneNodes()**).
- **ImageRecognitionARToolkit**
  Diese Klasse kapselt das ARToolkit sowie alle von diesem benötigten Parameter wie

threshold, confidence factor limit oder Größe des Kamerabildes. Bei der Initialisierung wird das Capturing der Kamera gestartet, die Linsenparameter gesetzt (siehe auch 3.1 Vermessung der Linsenparameter) und die Patterns geladen und registriert. Die wichtigste Methode dieser Klasse ist allerdings die Methode **detectMarkers(list<PatternInfo*>* patternInfoList, int* retCandidateMarkersCount)**. Diese Methode wird vom MemARyFrameListener aufgerufen und realisiert den Algorithmus zur Patternerkennung (siehe 2.3).

- **SceneNodeManager**
  Der SceneNodeManager übernimmt die Anpassung des Szenengraphen von Ogre3D an die aktuell sichtbaren Patterns. Dazu erzeugt er für jedes Pattern einen eigenen SceneNode, welchen er in einer Map speichert und ggf. sichtbar oder unsichtbar macht, je nachdem ob über die Kamera das entsprechende Pattern erkannt wurde oder nicht.

3 Vom Pattern zum AR-Erlebnis

3.1 Vermessung der Linsenparameter

Verschiedene Kameras besitzen unterschiedliche Linsenparameter. Die Berechnung der Transformationsmatrix anhand der aufgenommenen Bilder muss diese verschiedenen Linsenparameter berücksichtigen um korrekte Ergebnisse zu liefern. Die Vermessung der Linsenparameter erfolgt mit Hilfe eines Programms welches dem ARToolkit beiliegt. Zur Durchführung der Vermessung müssen verschiedene Muster ausgedruckt und im Kamerabild verschiedene Merkmale der Muster markiert werden. Nach mehrmaligem Wiederholen dieser Prozedur aus verschiedenen Betrachtungswinkeln und –entfernungen berechnen die benutzten Programme die Parameter der in der verwendeten Kamera verbauten Linse. Eine genaue Beschreibung und Dokumentation dieser Tools befindet sich auf der Homepage von ARToolkit (http://www.hitl.washington.edu/artoolkit/documentation/usercalibration.htm). Beim Abschluss der Vermessungsprozedur wird eine Datei erzeugt, die bei der Initialisierung der Bilderkennung dem Funktionsaufruf **arParamLoad(const char* filename, int num, ARParam* param, …)** übergeben werden muss.

3.2 Erzeugen der zu erkennenden Patterns

Die Patterns, die mit Hilfe des ARToolkit erkannt werden sollen müssen im Voraus erzeugt und bei der Initialisierung des ARToolkit eingelesen werden. Für das Erzeugen steht ebenfalls ein Programm zur Verfügung, welches aus dem Kamerabild automatisch das vorgehaltene Pattern extrahiert und in einer Datei abspeichert. Diese Datei muss danach zusammen mit dem entsprechenden 3D-Modell wie unter 2.2 Patternverwaltung mit XML beschrieben in die Datei patterns.xml eingetragen werden und wird bei der Initialisierung des ARToolkit über den Aufruf **arLoadPatt(const char* filename)** eingelesen. Die Patterns werden in den Dateien entzerrt gespeichert und zwar viermal jeweils um 90° gedreht und für jede Drehung jeweils dreimal für die Farben rot, grün und blau. Die Auflösung der Patterns ist 16x16 Bildpunkte, dies lässt sich aber im Quellcode von ARToolkit anpassen.

3.3 Erstellen der benötigten 3D Modelle mit Hilfe von 3D Studio Max

3.3.1 Low–Poly-Modelling allgemein

Für den Einsatz bei Spielen, ist es wichtig, die 3D-Daten schnell und interaktiv zu steuern und darzustellen. Um dies zu erreichen, ist eine Vereinfachung der Daten oberste Priorität. Geringe

Elementzahlen bedeuten eine effizientere Ausbeutung hinsichtlich der Geschwindigkeit vor allem bei der Darstellung. Deshalb spricht man in diesen Bereichen auch vom sog. Low – Poly – Modelling.

Dies bedeutet, dass man versucht, mit möglichst wenigen Informationen (Polygonen) eine möglichst hohe Qualität der Darstellung zu erreichen. Dies wird in der Regel durch eine geschickte Wahl der Materialien unterstützt. Allerdings bedeutet dies auch eine ständige Gratwanderung zwischen Qualität der Darstellung und der geforderten Performance.

Grundsätzlich gibt es zwei Arten, ein 3D-Objekt mit wenigen Polygonen zu erstellen.

Man kann ein Objekt ohne Beachtung der Polygonzahlen modellieren und dann anschließend durch Anwenden von Modifikatoren eine Reduzierung der Polygonzahlen erreichen oder man modelliert das 3D-Objekt von Anfang an als Low – Poly.

Modifikatoren, die eine Reduktion der Polygonanzahl bewirken, sind zum Beispiel der Optimieren- und der MultiRes – Modifikator.

Der Optimieren – Modifikator zum Beispiel vereinfacht vorhandene Geometrien durch die Reduktion von Flächen. Hierbei wird versucht, das ursprüngliche Aussehen des Objektes beizubehalten.

Auch bei dem MultiRes – Modifikator geht es darum, Geometrien zur Speicherreduzierung zu vereinfachen. Allerdings gibt es hier detailliertere Möglichkeiten, in die Geometrie einzugreifen. Es lassen sich zum Beispiel die genaue Scheitelpunktanzahl bestimmen oder diese als Prozentwert angeben. Als Scheitelpunkt werden alle Eckpunkte der Dreiecksflächen bezeichnet, aus denen ein Objekt zusammengesetzt ist.

3.3.2 Box–Modellierung

Bei der Box – Modellierung hingegen ist ein Quader, die Grundlage komplexer Modelle. Dieser Quader wird so lange modifiziert, bis das resultierende Objekt die gewünschte Form besitzt.

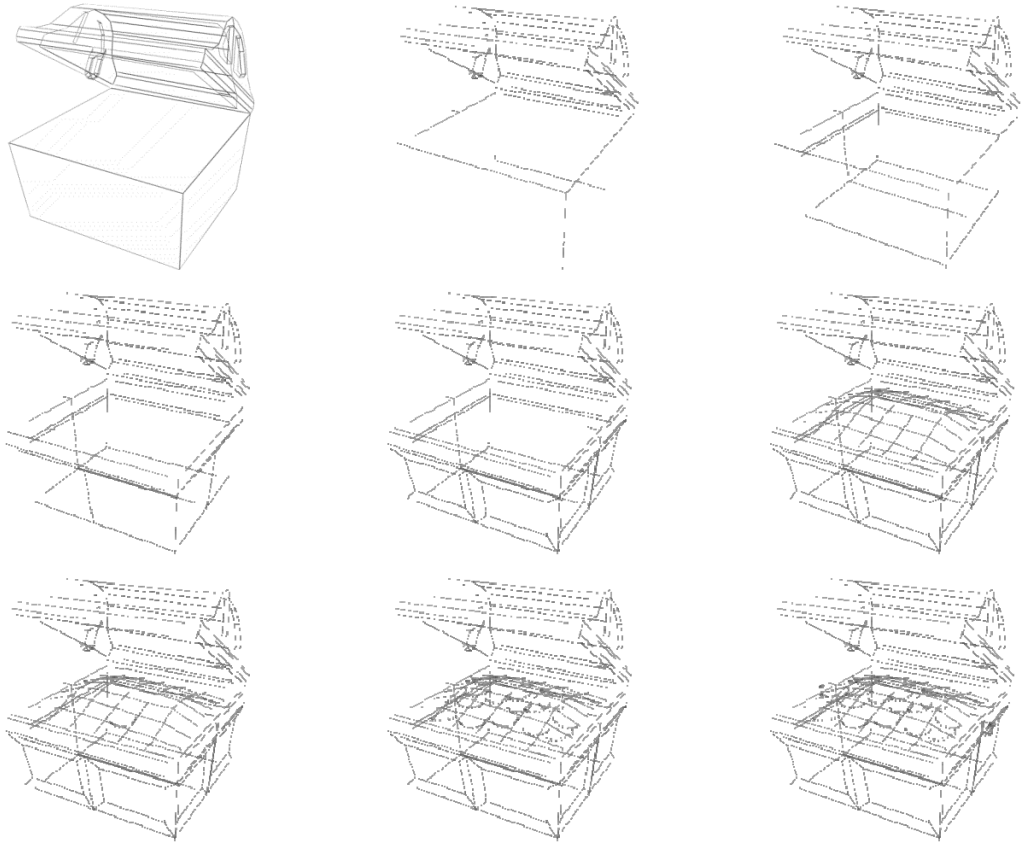
Als ersten Schritt wird dieses Basisobjekt in ein Editable Poly oder in ein Editable Mesh konvertiert. Dieser Schritt ist Grundlage für alle weiteren Schritte der Modellierung, um die Modifikationen auf den Vertex-, Edge-, Face- und Polgonebene durchführen zu können. Für die Modellierung in 3ds max hat sich bewährt, sich

- die Statistiken beim Modellieren anzeigen zu lassen, um die Polygonanzahl immer überwachen zu können
- die Ansicht „Edged Faces“, um den Verlauf der Kanten im Blick zu haben.

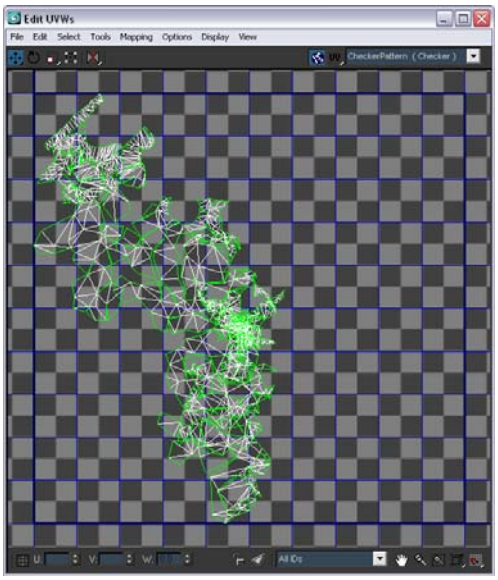
Bei der Low – Poly – Modellierung nehmen sog. **Glättungsgruppen** eine wichtige Rolle ein, da sie ein weiteres Merkmal zur Bestimmung des Erscheinungsbildes von Oberflächen sind. Haben zwei aneinander liegende Flächen die gleiche Nummer bzgl. Der Glättungsgruppe, so wird der Übergang zwischen den beiden Flächen als glatte Oberfläche dargestellt. Sind die Nummern dieser Flächen unterschiedlich, so sieht man die Kante.

Tests mit dem Ogre3D-Exporter haben gezeigt, dass es besser ist, die einzelnen Baugruppen des fertig modellierten 3D-Objekts zu einem einzigen Objekt zusammenzuschmelzen.

Im Hinblick auf die anschließende Texturierung des Objekts ist es sinnvoll, **Materialindizes** innerhalb des Objektes für verschiedene Baugruppen zu vergeben.



3.3.3 Unwrapping/ Mapping



Material bestimmen das Erscheinungsbild der einzelnen geometrischen Objekte. Ein gelunges Material unterstützt die Form und hebt diese hervor. Ein Material kann aber auch die Geometrie dominieren und von dieser ablenken. Bei der Low – Poly – Modellierung nehmen Texturen eine wichtige Rolle ein, da in der Regel bei der Modellierung auf Details verzichten wird. Diese werden dem Objekt dann über die Textur hinzugefügt.

Mapping ist der gängige Begriff für die Belegung von 3d-Oberflächen mit einer Textur bzw. einem Material.

Man bezeichnet Mapping-Koordinaten als UV oder UVW-Koordinaten. Diese Buchstaben beziehen sich auf die Objektraumkoordinaten im Gegensatz zu den xyz-Koordinaten, mit denen die gesamte Szene beschrieben wird.

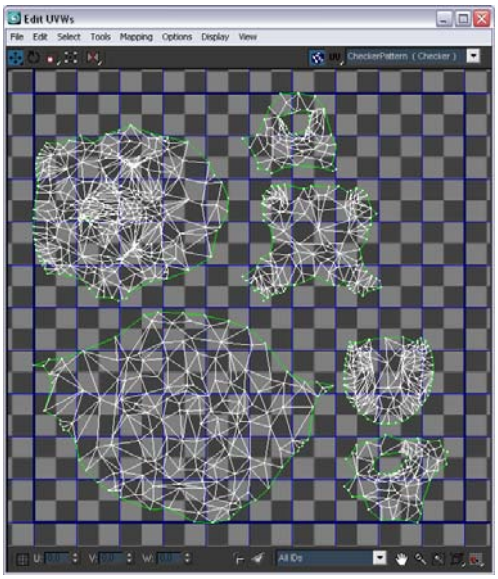
Bei jedem Mapping ist es wichtig zu wissen, in welcher Richtung und Größe eine Textur auf das Objekt aufgebracht werden soll. Das Aussehen des Materials wird im Materialeditor festgelegt, die Art der Projektion muss aber direkt in der Geometrie eingestellt werden. Fehlen bei einem Objekt beim Rendern die Mapping – Koordinaten, werden sie so weit wie möglich automatisch nachträglich generiert, damit texturierte Materialien richtig dargestellt werden können.

Nicht immer bewirken die standardmäßig erstellten Mapping – Koordinaten das gewünschte Ergebnis. Der **UVW – Map – Modifikator** bietet die Möglichkeit, diese Koordinaten beliebig auf ein

Objekt einzustellen. Dieser Modifikator unterscheidet mehrere Möglichkeiten, wie Maps auf Objekte projiziert werden können. Die unterschiedlichen Projektionsarten, die im Parameterrollout eingestellt werden können, können sowohl für das gesamte Objekt, als auch für einzelne Flächen eines Objektes eingestellt werden.

Die wichtigesten Projektionsarten dabei sind:

- Planar, die von einer Ebene aus, auf die Objekte projiziert
- Zylindirsch, die die Textur um einen Zylinder wickelt
- Kugelförmig, wie bei einem Globus
- Quader, eine ebene Projektion aus den sechs im Raum aufeinander senkrecht stehenden Richtungen
- Fläche, hierbei wird auf jede Fläche einzeln projiziert
- XYZ in UVW – gilt für prozedurale Maps. Die 3D – Koordinaten der Map werden auf die Oberfläche eines Objekts projiziert, so dass eine Veränderung des Objekts scheinbar die Textur nicht beeinflusst, Die Textur scheint, auf der Oberfläche zu kleben.



Nachdem man nun für einige Flächen oder sämtliche Flächen eines Objekts die Projektionsart gewählt hat, kann man über die Option „Edit“ die UVW’s weiter bearbeiten. Ziel dieser Bearbeitung ist die Anpassung der Größe und die genauere Anordnung der Flächen, die auf die Flächen des Objekts gemappt werden sollen hinsichtlich der zu erstellenden Textur. Eine gut durchdachte Anordnung erleichert die spätere Texturierung erheblich. Über das Checker – Pattern lässt sich das voreingestellte Schachbrettmuster direkt im Viewport anzeigen und die Auswirkung jeder Änderung der Texturkoordinaten lässt sich direkt am Objekt überprüfen. Man sollte bei den Änderungen darauf achten, dass die einzelnen Flächen des Schachbretts im Viewport ebenfalls quadratisch sind, um zu vermeiden, dass dir Textur bei der Zuweisung an die Koordinaten des Objekts gestaucht oder gestreckt werden.

Nachdem die Bearbeitung der UVW – Koordinaten abgeschlossen ist, rendert man das UVW – Template heraus. Dieses Template kann man als Textur abspeichern und dem Objekt zuweisen.

3.3.4 Texturierung

Da das herausgerenderte Template lediglich die einzelnen Kanten des Objektes als Linien darstellt, muss man nun die Textur gestalten. In diesem Projekt verwendeten wir Adobe Photoshop CS3, da es die Vorzüge der Ebenenmasken aufweist. Seamless – Texturen erleicherten erheblich die Texturierung. Diese sind dadurch gekennzeichnet, dass ein Aneinanderreihung dieser Textur keine Bildgrenzen zwischen den einzelnen Texturen sichtbar werden lässt. Bei Tapeten lässt sich im Normalfall auch dieser Umstand erkennen.

Durch die zuvor beschriebenen Arbeitsschritte Modellierung, Mapping und Texturierung gelangt man zum fertigen 3D-Objekt.

FAZIT:

- Vereinfachungsmodifikatoren sind ein mächtiges Werkzeug. Bei Low – Poly – Modellierung gibt man allerdings die Kontrolle über die Kanten im Polygonnetz durch Einsatz dieser Modifikatoren aus den Händen. Dadurch erschwert sich das gezielte Unwrapping/ Mapping.
- Regelmäßige Überprüfung des Gittermodells bei der Modellierung und besonderes Augenmerk auf die Vertexanzahl markierter Vertizes, um Überlappungen von Flächen zu vermeiden
- Details eines Objektes sollten durch die Textur dem Objekt hinzugefügt werden (siehe Münzen in der Schatzkiste) um die Anzahl der Polygone möglichst gering zu halten
- Die Textur sollte die Bildmaße 512 Pixel x 512 Pixel nicht überschreiten. In vielen Spielen wird sogar eine Maximalgröße von 256x256 vorgeschrieben.

3.3.5 Exportieren der 3D Modelle für Ogre3D

Bevor man ein Modell exportiert, sind folgende Punkte zu beachten:

- Um eine genaue Positionierung eines Objekts zu ermöglichen, ist es erforderlich, den Pivot des Objektes auf den Koordinatennullpunkt zu verschieben
- Sämtliche Modelle sollen eine vorher definierte räumliche Ausdehnung nicht überschreiten
- Ein Modell darf nur aus einem Objekt bestehen und dieses Objekt darf keine Löcher im Polygonnetz aufweisen
- Die Transformationsmatrix eines Objektes muss mittels XForm - Modifikator zurückgesetzt werden.