

# Ein zustandsbasiertes, ereignisgesteuertes Virtual Reality Framework

Matthias Biedermann, Oliver Abert, Jonas Fleer, Stefan Müller

Institut für Computervisualistik

Universität Koblenz-Landau

56070 Koblenz

Tel.: +49 (0)261 287 2794

Fax: +49 (0)261 287 2735

E-Mail: mbmann@uni-koblenz.de

**Abstract:** Wir präsentieren eine Systemarchitektur, die speziell für Anwendungen im Bereich des VR-Edutainment geeignet ist. Im Gegensatz zu konventionellen VR-Systemen basiert unsere Systemarchitektur auf einem Statemanagement, welches über eine Konfigurationsdatei konkreten Bedürfnissen angepasst werden kann. Darüber hinaus bietet unser auf OpenGL basierendes Renderingsystem die Möglichkeit, Applikationen problemlos auch in Stereo zu präsentieren. Die einzelnen Komponenten des Systems kommunizieren mit Hilfe eines flexiblen Eventmanagers miteinander, der ebenfalls ohne weitere Änderungen für andere Szenarien eingesetzt werden kann. Das System wurde im Rahmen eines Projektpraktikums durch die prototypische Umsetzung der „Virtuellen Erlebniswelt Mittelrheintal“ auf seine Leistungsfähigkeit hin getestet.

**Stichworte:** Framework, Virtual Reality, Edutainment, Statechart, Eventmanagement

## 1 Einleitung

Kulturelles Wissen wird den Menschen meistens in Büchern oder Erzählungen näher gebracht, die manchmal durch alte Fotos oder Illustrationen bereichert werden. Obwohl man sich dabei oft in die damalige Zeit hineinversetzen kann, bleiben dem Zuhörer insbesondere Alltagsleben, Architektur oder Kleidung oftmals verschlossen. Es ist also wünschenswert, das Publikum in die Vergangenheit „eintauchen“ zu lassen und es in beliebige Orte und Zeiten zu versetzen, so dass es diese hautnah selbst erleben kann. Außerdem behält der Mensch das, was er selbst *erfährt* wesentlich besser, als wenn er es nur liest oder hört.

Durch geeignete Techniken der Computergraphik lassen sich virtuelle Welten erstellen, in denen sich der Betrachter frei bewegen und so spielerisch kulturelles Wissen aneignen kann, in dem er selbst Teil der erzählten Welt wird. Die flexible Realisierung und Kombination der Möglichkeiten ist Gegenstand dieses Projektes.

## 2 Das Gesamtsystem

Das hier vorgestellte System wurde mit dem Ziel entwickelt, VR-Szenarien aus dem Edutainmentbereich mit nicht-linearen Handlungssträngen möglichst vollständig und flexibel verarbeiten zu können. Dazu gehört neben den eigentlichen Inhalten, die in unserem Fall exemplarisch in der „Virtuellen Erlebniswelt Mittelrheintal“ umgesetzt wurden, auch eine Plattform zur Eingabe, Verarbeitung und Ausgabe von VR-typischen Daten.

Üblicherweise basieren Abläufe für VR-Anwendungen aus dem Edutainmentbereich auf vorher erstellten Storyboards, die die gewünschte Handlung wiedergeben. In unserem Fall sollte aber zusätzlich die Möglichkeit geschaffen werden, unterschiedliche Szenarien ohne signifikante Eingriffe in das System umsetzen zu können. Zu diesem Zweck wird eine vorher erstellte, zustandsbasierte Repräsentation der Handlung in Form von XML-Statecharts interpretiert und so das System in neue Zustände überführt.

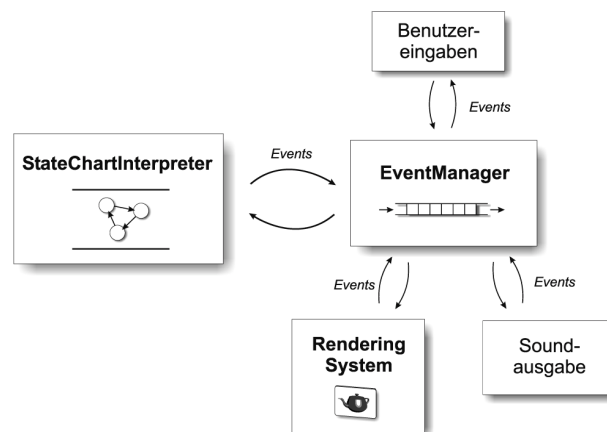


Abbildung 1: Überblick über das Gesamtsystem

Insbesondere diese zentrale Zustandssteuerung benötigt zudem eine Möglichkeit, mit anderen Komponenten des Gesamtsystems zu kommunizieren. Der hierfür verwendete Eventmanager koordiniert diesen Informationsaustausch als einen Fluss von Ereignissen. Dieser Eventmanager arbeitet in Verbindung mit dem StateChartInterpreter, um die Handlungsstrukturen in konkrete Programmaktionen umsetzen zu können.

Ein solches VR-System benötigt zudem eine Reihe von Ausgaben, im Wesentlichen also die Darstellung des Szenarios auf (Stereo-)Bildschirmen sowie die Wiedergabe entsprechender Sounds. Diese Bestandteile, die neben der Eingabe die direkte Schnittstelle zum Benutzer darstellen, verbinden die internen Verarbeitungssysteme (Eventmanager und StateChartInterpreter) „mit der Außenwelt“; die Abläufe selbst werden dabei wiederum durch Ereignisse gesteuert.

Nach diesem Überblick (siehe Abbildung 1) sollen im Folgenden die drei wesentlichen Bestandteile Renderingsystem, Statemanagement sowie die Eventsteuerung detaillierter vorgestellt werden.

### **3 OpenSG Renderer**

Nachdem wir verschiedene Szenegraphsysteme hinsichtlich ihrer Verwendbarkeit für unser Vorhaben untersucht haben, entschieden wir uns für OpenSG [OSG2003]. Der ausschlaggebende Grund dafür war, dass dieses System eine einfache und leistungsstarke API bietet, um Applikationen auch in Stereo auf einem heterogenen Cluster zu rendern.

#### **3.1 Clustering**

Für größere Stereo-Applikationen ist es empfehlenswert, einen Verbund aus mindestens drei Rechnern zu verwenden. Je ein Rechner (Renderserver) berechnet dabei das Bild für linkes bzw. rechtes Auge, während der dritte Rechner (Client) die Synchronisation der ersten beiden übernimmt und Simulationsberechnungen der Applikation durchführt. Bei Bedarf kann der Cluster auch weiter skaliert werden, so dass sich beispielsweise mehrere Rechner die Erzeugung eines Bildes teilen. Die Renderserver selbst werden nur über Änderungen des Szenegraphen informiert und stellen diese dar. Um die in jedem Frame zu übertragende Datenmenge zu minimieren, geben so genannte „change lists“ nur die Veränderungen über das Netzwerk an alle registrierten Renderserver weiter.

#### **3.2 Kollisionserkennung**

Jede virtuelle Welt benötigt eine Möglichkeit zur Kollisionserkennung von bewegten Objekten, wobei die Kollision des Benutzers (der Kamera) mit Objekten, wie z.B. Wänden oder Avataren eine wichtige Rolle spielt. Darüber hinaus muss für eine korrekte Bewegung ständig die Höhe des Benutzers überprüft und gegebenenfalls angepasst werden. Um diese Forderungen zu erfüllen, wird in unserem System sowohl ein Strahl von der aktuellen Position in Bewegungsrichtung, als auch senkrecht nach unten für die Höhenanpassung verfolgt, so dass gegebenenfalls eine neue gültige Position bestimmt werden kann.

Problematisch ist bei diesem Vorgehen jedoch, dass diese Strahlen prinzipiell *in jedem Frame* verfolgt werden müssen, was bereits bei einer verhältnismäßig geringen Anzahl von Polygonen durchaus zu Performanzproblemen führen kann. Aus diesem Grund haben wir für alle relevanten Modelle zusätzliche Kollisionsobjekte erstellt, die das Original mit möglichst wenigen Flächen annähern. In der Konfigurationsdatei werden diese Kollisionsmodelle als solche markiert und in einem eigenen Teil des Szenegraphen verarbeitet, der ausschließlich dem Strahltest dient und daher nicht dargestellt wird.

#### **3.3 Laden von 3D-Modellen**

Als Dateiformat für den Datenaustausch haben wir uns für VRML [IEC14772-1] entschieden, um die Daten aus der Modellierungssoftware ohne Umweg über einen zusätzlichen Konverter in OpenSG laden zu können. Da VRML-Dateien aufgrund ihrer Größe bei komplexeren Modellen zu erheblichen Verzögerungen beim Laden während des Startvorgangs führen, wurde unser Lademodul um die Möglichkeit einer automatischen Generierung einer Binärversion erweitert, die anschließend allen weiteren Ladevorgängen direkt zur Verfügung steht.

### 3.4 Szenegraph-Abstraktionsebene

Sollen im Szenegraph neue Knoten hinzugefügt, modifiziert oder gelöscht werden, benötigt man verhältnismäßig viel Code. Um diesen zusätzlichen Aufwand zu reduzieren, haben wir eine eigene Szenegraph-Klasse entwickelt, die häufig benötigte Funktionen von OpenSG in eigenen Methoden kapselt und zur Verfügung stellt. Mit Hilfe dieser Abstraktion können auf einfache Art und Weise komplette Szenen, wie sie in der Konfigurationsdatei definiert sind, eingelesen oder auch wieder entfernt werden; eventuell zugehörige Kollisionsmodelle werden dabei automatisch mit verwaltet.

## 4 StateChartInterpreter

Im Folgenden beschreiben wir die Implementation eines Interpreters für in XML [W3C1996] annotierte Statecharts, der dem Benutzer die Interaktion mit virtuellen Welten ermöglicht und ansprechende, nicht-lineare Storyboards realisiert.

### 4.1 Naiver Lösungsversuch

Ein erster naiver Ansatz wäre, Handlungsabläufe mit in den Programmquelltext zu kodieren. Dieses Vorgehen ist unter Performanzaspekten sicherlich eine effiziente Lösung und für kleinere System durchaus auch vertretbar. Ein Beispiel wäre folgender Pseudocode für die Interaktion mit einem Avatar:

```
if ( talkingWith( Kartenverkaufferin ) )
    if ( pressed( BUTTON_X ) ) {
        startFlight( Ehrenbreitstein );
        playSound( "windpfeifen.wav", user_position );
        setNewPosition( Ehrenbreitstein );
    }
else if ( standingAt( Ehrenbreitstein ) )
    ...
```

Es ist offensichtlich, dass dies nicht die Lösung für ein größeres, flexibles System darstellen kann. Zum einen vermischen sich Storyboard und Programmierung, so dass die Umsetzung der Handlung praktisch nur von Programmierern durchführbar ist. Zum anderen wird der Code für das Programm *und* die Story schlechter wartbar und muss darüber hinaus bei jeder Änderung der Handlung neu kompiliert werden.

### 4.2 Systembeschreibung mit Statecharts

*Reaktive Systeme* sind Systeme, die in ständiger Interaktion (durch Benutzereingaben, Zeitbedingungen etc.) mit der Umgebung und sich selbst stehen. Zudem zeigen sie auf Ereignisse in Abhängigkeit ihres aktuellen Zustands Reaktionen, die wiederum Ereignisse sind. *Statecharts*, deren grafische Notation im UML-Standard [RuJaBo1999] definiert ist, sind ein in

der Softwaretechnik angewandtes Modellierungskonzept, um das Verhalten von solchen Systemen zu beschreiben.

In Abbildung 2 modellieren wir beispielsweise das Betreten des Benutzers eines Thronsaals von einem Schlossvorplatz aus. Es laufen dabei drei Sub-Statecharts parallel, was durch eine gestrichelte Trennungslinie gekennzeichnet wird. Der erste modelliert die Position des Benutzers (anfangs noch auf dem Schlossvorplatz), der zweite die Beziehung zwischen dem Benutzer und dem Schlüssel für die Eingangstür des Schlosses (zu Beginn habe der Benutzer keinen Schlüssel) und der dritte den Untergrund, auf dem sich der Benutzer bewegt (der Schlossvorplatz, auf dem der Benutzer anfangs steht, sei mit Kiesel bedeckt).

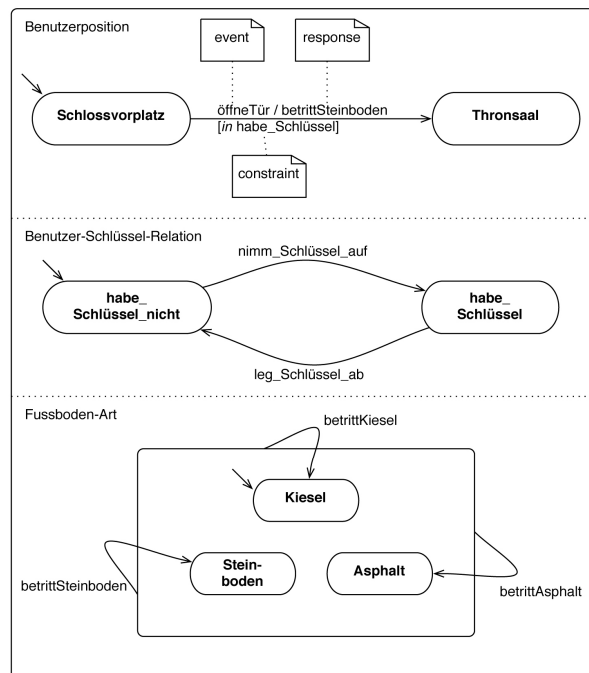


Abbildung 2: Statechart für eine kleine Beispielwelt

Die *Transitionen* (Zustandsübergänge) der *States* (Zustände) werden ausgeführt, wenn das der Transition annotierte *Event* (Ereignis) von außen oder anderen Stellen des Statecharts gesendet wird. Letzteres sind *Responses* (Antworten), die, an Transitionen annotiert, sozusagen als eine Antwort nach einer Durchführung einer Transition gesendet werden, über die die Sub-Statecharts miteinander kommunizieren können. Im vorangehenden Beispiel teilt der Statechart für die Benutzerposition dem Statechart für den Untergrund mit, in den Zustand des Steinbodens zu wechseln, sobald der Thronsaal betreten wird (da das externe Event `öffneTür` z.B. vom Gamepad gesendet wurde).

Zusätzlich können die Transitionen noch von beliebig vielen *Constraints* (Einschränkungen), die in eckigen Klammern annotiert werden, abhängig gemacht werden. Dadurch wird der Wechsel in einen anderen State (und das Senden von Responses) nur vollzogen, wenn sich die in den Constraints genannten States in der Menge der aktiven States befinden. In unserem Beispiel wird der Benutzer also nur in den Thronsaal kommen können, wenn er zuvor den dazugehörigen

Schlüssel aufgenommen hat (im Beispiel durch die Transition `nimm_Schlüssel_auf` vermerkt).

### 4.3 Definition von Statecharts in XML

Da wir die Statecharts wegen der in Abschnitt 4.1 beschriebenen Probleme nicht mit dem Quelltext vermischen wollen, sich andererseits aber eine graphische Notation der Statecharts kaum direkt weiterverarbeiten lässt, müssen wir sie in einem geeigneten, von einem Parser verarbeitbaren Datenformat definieren. Wir haben uns für XML entschieden, da es sowohl leicht zu notieren ist, als auch aufgrund der bewährten und als OpenSource verfügbaren Parser- und Validierungsbibliotheken gegenüber einer eigenen Syntax vorzuziehen ist.

Wir haben eine eigene Document Type Description (DTD) entworfen, mit deren Hilfe unsere Statechart-Definitionsdateien auf ihre Gültigkeit hin geprüft werden können. Die verschiedenen Arten von States (elementar, parallel, hierarchisch) und die Transitionen sind als eigene XML-Tags definiert; Events, Responses, Constraints und Aktionen hingegen als Attribute. Hier ein Beispiel für einen gemäß unserer DTD verfassten Statechart, der dem Fallbeispiel aus Abbildung 2 entspricht:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE VRSCI SYSTEM "vrstatechart.dtd">
<xorblob name="root">
  <andblob name="Mittelrheintal">
    <xorblob name="Benutzerposition">
      <elemblob name="Schlossvorplatz">
        <transition name="öffneTür" sink="Thronsaal" load="ThronsaalScene"
          send="betrittSteinboden" constraints="habe_Schlüssel"/>
      </elemblob>
      <elemblob name="Thronsaal" soundenv="hall" />
    </xorblob>
  ...

```

### 4.4 Implementierung eines StateChartInterpreters

Statecharts werden üblicherweise nur zur Modellierung des Systemverhaltens während des Softwareentwurfs benutzt. Das modellierte System wird daraufhin im Rahmen dieser Beschreibung implementiert. Wir verfolgen hingegen das Ziel, das Verhalten (also die Handlung und Umgebung) der VR-Anwendung auch nachträglich ändern zu können. Die Zustände und ihre Übergänge sollen nicht statisch implementiert werden (siehe Abschnitt 4.1), sondern so, dass sich das System lediglich durch Ändern einer Konfigurationsdatei vollkommen anders verhalten kann. Dazu ist es naheliegend, eine abstrakte Zustandsübergangs-Beschreibung zu entwerfen, deren konkrete Instanzen dann alle möglichen Beschreibungen dieser Art abdecken können, sowie einen entsprechenden Interpreter zur Umsetzung der Informationen zu implementieren. Wir haben dabei die zuvor genannten Konzepte von Statecharts (States, Transitionen, Events, Responses, Parallelität, Hierarchisierung, Constraints) als Grundlage für eine Implementation herangezogen.

Als zusätzliches Konzept zum Statechart-„Standard“ wollen wir *Aktionen* einführen, die States und Transitionen mit Programmlogik ausstatten. Konkret definieren wir eine Menge von Attributen an unsere Beschreibung von States und Transitionen, die dann in unserer Applikation Funktionen zuzuordnen sind, also beispielsweise dem Nachladen von Szenen etc. Modellieren wir ein Storyboard, so weisen wir in unserem Statechart diesen Attributen konkrete Werte zu, die dann in der Applikation als Parameter für die zugeordneten Funktionen dienen.

Der StateChartInterpreter liest zum Systemstart die im vorherigen Abschnitt beschriebene XML-Datei, baut daraus eine interne Repräsentation des definierten Statecharts als zyklischen, gerichteten Graph auf und aktiviert danach den initialen State. Während des Programmablaufs werden vom EventManager Transitionen durch Events ausgelöst und verarbeitet. Auf diese ereignisgesteuerten Abläufe wird im Zusammenhang mit dem EventManager in Abschnitt 5 noch detaillierter eingegangen.

#### **4.5 Verbesserungsmöglichkeiten**

Der hier präsentierte StateChartInterpreter ist nicht auf konkrete Modelle oder Storyboards, jedoch stark auf die Architektur unseres Systems und dabei insbesondere des EventManagers festgelegt. Eine weitere Einschränkung unserer Implementation ist die Festlegung auf unsere DTD, die bei einer Erweiterung der möglichen Programmlogik (neue Aktionen für States und Transitionen) zusammen mit dem StateChartInterpreter angepasst werden müsste.

Ein idealer StateChartInterpreter würde dagegen ein den Statechart-Definitionen vorgeschaltetes Statechart-*Schema* einlesen, das eine Definition für beliebige Statecharts mit beliebigen Aktionen sowie eine Abbildung dieser auf Funktionen des Systems enthält. Wo und wie das Tupel (Attributname, Attributwert) auf das Tupel (Objektname.Funktionsname, Parameter) abgebildet wird, ist dabei jedoch noch fraglich. Mit Hilfe interpretierter Sprachen wäre eine solche Ausweitung der Anwendungsdomäne für eine Verwendung in jedem dynamischen reaktiven System einfacher möglich, als in einer Programmiersprache wie z.B. C++.

Eine allgemeinere Implementation, wie sie in [Eb1993] beschrieben ist, existiert als Baustein des Meta-CASE-Systems „KOGGE“ (Koblenzer Generator für Graphische Entwurfsumgebungen), das am Institut für Softwaretechnik an der Universität Koblenz-Landau entwickelt wurde. Diese war für unser Projekt wegen ihres Umfangs jedoch nicht verwendbar.

## **5 EventManager**

Wie bereits angedeutet wurde, kommunizieren die einzelnen Komponenten unseres VR-Systems mit Hilfe von *Events* (Ereignisse). Diese werden sowohl durch externe Einflüsse (z.B. Benutzereingaben), als auch durch interne Vorgänge (siehe Abschnitt 4) erzeugt und an geeigneter Stelle verarbeitet. Dabei ist es natürlich wünschenswert, dass das Eventmanagement möglichst wenige (einschränkende) Anforderungen an die gesamte Architektur stellt, um ausreichende Flexibilität für verschiedene Anwendungsgebiete zu gewährleisten. Hier bietet der Ansatz der Events eine ideale Lösung. Miteinander kommunizierende Komponenten erzeugen

Eventobjekte und übergeben sie dem Eventmanagementsystem, das für deren Weiterverarbeitung sorgt. Erzeuger und Verbraucher müssen bei diesem Konzept keine Kenntnis voneinander besitzen, was insbesondere für dynamische Umgebungen wie VR-Anwendungen wichtig ist. So können anhand des Storyboards, d.h. in unserem Framework auf Veranlassung des StateChartInterpreters, zur Laufzeit die einzelnen Komponenten *unabhängig* voneinander und ohne direkte Verbindung verwendet werden.

Im folgenden Abschnitt sollen die wesentlichen Konzepte des hier umgesetzten Eventmanagementsystems beschrieben werden. Dabei geht es zum einen um die Details des Eventmanagers selbst, zum anderen auch um die Integration und das Zusammenspiel mit den anderen Komponenten unseres VR-Systems.

Neben den Performanzanforderungen, die im Rahmen von VR-Anwendungen an eine zentrale Komponente wie den EventManager (EM) gestellt werden, ist das Gesamtsystem vor dem Hintergrund konzipiert worden, leicht für verschiedene VR-Welten einsetzbar zu sein. Dies führt zu der Notwendigkeit einer möglichst *kleinen Schnittstelle* zum restlichen VR-System, um die Komponenten aus softwaretechnischer Sicht nicht zu stark voneinander anhängig zu machen; der EM nimmt dabei „nur“ die Rolle einer *Vermittlungsstelle* ein. Zum anderen sollte er mit den potentiell sehr unterschiedlichen Anforderungen anderer Szenarien ohne Anpassungen umgehen können. Eine *Hierarchie von Eventklassen*, die von einer gemeinsamen und vom EM ausschließlich verarbeiteten Basisklasse ausgeht, repräsentiert dabei die individuellen Bedürfnisse der jeweiligen Anwendung.

## 5.1 Schnittstellen zum VR-System

Der EM spielt zwar eine zentrale Rolle für die Kommunikation aller Komponenten, soll aber andererseits möglichst unabhängig von diesen Bestandteilen arbeiten; es werden dabei also mindestens folgende Funktionalitäten benötigt:

- *Verarbeiten von Ereignissen*  
Hierbei wird die Verarbeitung der eingetroffenen Events veranlasst.
- *Registrieren von Empfängern für einen Eventtyp*  
Die semantische Verarbeitung der Events geschieht erst beim Verbraucher (EventListener), der sich für ein oder mehrere Ereignistypen registrieren kann. Dies geschieht im Wesentlichen wiederum mit Hilfe von Ereignissen, die vom StateChartInterpreter auf Basis des Storyboards und/oder anderen Einflüssen erzeugt werden. Die Deregistrierung erfolgt dabei analog.
- *Übergeben eines Eventobjektes an den EM*  
Diese zentrale Funktion ermöglicht es den verschiedenen Komponenten des Systems, mit Hilfe beliebiger Ereignisobjekte aus der Eventhierarchie mit anderen Modulen zu kommunizieren.

## 5.2 Die Ereignis-Hierarchie

Aus den vorherigen Ausführungen sollte klar geworden sein, dass es je nach Einsatzgebiet eine Vielzahl von verschiedenen Ereignissen gibt. Zudem können sich diese Events in ihrer Wichtigkeit unterscheiden, d.h. es gibt Klassen von Ereignissen, die schnellstmöglich übermittelt werden müssen, während andere keine derartige Bevorzugung benötigen. Um die Schnittstelle zum EM von den konkreten Ereignissen zu abstrahieren, wird lediglich die gemeinsame (abstrakte) Basisklasse aller Events verarbeitet, was sich mit den Möglichkeiten objektorientierter Programmierung leicht umsetzen lässt.

Die priorisierte Behandlung von Ereignissen geschieht in unserem Framework mit Hilfe eines zweistufigen Systems, das Attribute jedes Eventobjektes vergleicht. Als erstes (Sortier-) kriterium sind vier verschiedene Prioritätsstufen global definiert, die die Events zunächst grob einteilen. Zusätzlich ermöglicht ein Zeitstempel auf Basis der Systemzeit die Unterscheidung von Events gleicher Prioritätsstufe bezüglich ihres „Alters“. Diese Grob- und Feinsortierung ermöglicht dem EM eine ausreichend gute und gleichzeitig sehr effiziente Bevorzugung wichtiger Ereignisse.

## 5.3 Die interne Umsetzung

Wie aus Abbildung 3 hervorgeht, besteht der EM im Wesentlichen aus zwei Komponenten: der EventQueue und einer EventListenerList. Während erstere eine prioritätsbasierte Warteschlange für die Ereignisobjekte darstellt, enthält die Liste die Empfänger der verschiedenen Eventtypen, die unmittelbar mit der Eventhierarchie korrespondieren. Für jede Eventklasse wird eine (anfangs leere) Liste von EventListenern angelegt. Zur Laufzeit können die gewünschten EventListener-Objekte mit der in Abschnitt 5.1 beschriebenen Registrierungsfunctionalität hinzugefügt werden. Dadurch können den zu verarbeitenden Ereignissen nicht nur ein, sondern *beliebig viele* Empfänger zugeordnet werden; es wäre sogar möglich, dass sich Empfänger mehrfach für einen Eventtyp registrieren, um beim Eintreten eines solchen Ereignisses mehrere Aktionen auszuführen.

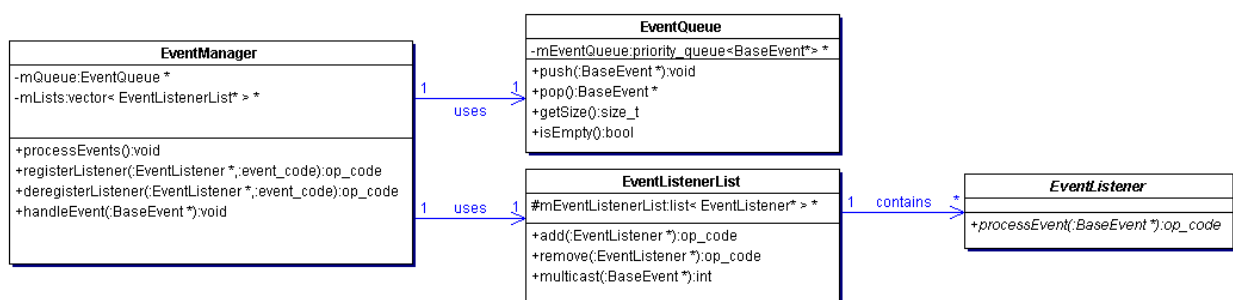


Abbildung 3: Klassendiagramm des EM-Kernsystems (vereinfacht)

Die bereits mehrfach erwähnten EventListener stellen schließlich die Umsetzung der Funktionalität zur Behandlung der Ereignisse dar. Wie dieser Verbraucher letztlich auf die Eventobjekte reagiert, also z.B. selbst wieder Events auslöst, diese an andere Empfänger (über

den EM) weiterleitet usw., ist für den EM nicht relevant, da die Schnittstelle ebenfalls durch Verwendung einer gemeinsamen Basisklasse konstant bleibt.

## 6 Ergebnisse

Im nun folgenden Abschnitt wird auf die praktischen Erfahrungen und Ergebnisse mit unserem System eingegangen. Dabei wird auch auf die exemplarische Umsetzung der VR-Welt eingegangen.

Sowohl die Systemarchitektur als auch das Anwendungsbeispiel „Virtuelle Erlebniswelt Mittelrheintal“ sind im Rahmen eines Projektpraktikums an der Universität Koblenz-Landau erstellt worden, an dem über einen Zeitraum von sechs Monaten 27 Studierende teilgenommen haben. Ziel war es, eine interaktive Erlebniswelt zu schaffen, in der der Benutzer frei durch eine Umgebung navigieren kann und dabei von einer Reihe von Avataren begleitet wird. Diese informieren ihn zu Geschichte und Kultur der Region, die 2002 zum UNESCO Weltkulturerbe erklärt wurde.



Abbildung 4: Avatar (Ballonfahlerin) am Deutschen Eck bietet Flug zur Festung Ehrenbreitstein an

Wie in Abbildung 4 dargestellt ist, startet der Benutzer am Deutschen Eck in Koblenz und kann mit einer Ballonfahlerin Ausflüge zu Zielen am Mittelrheintal unternehmen. Verschiedene Zustände und Zustandsübergänge des StateChartInterpreters, die das gesamte Storyboard realisieren, modellieren so das Verhalten der Avatare, sorgen für Einblendungen von Hintergrundinformationen auf einem externen Monitor, steuern Geräusche, Animationen und vieles mehr. Alle Modelle, die sich über ein Gebiet von Koblenz bis zur Loreley erstrecken, wurden eigens für das Projekt erstellt. Dabei wurden neben markanten Burgen am Rhein insbesondere das Deutsche Eck und die Festung Ehrenbreitstein in Koblenz sowie die Marksburg detailliert nachgebildet. Diese Stationen können vom Benutzer besucht und eigenständig erforscht werden. (siehe Abbildungen 5 bis 8).

Während diese Modelle von Grund auf modelliert wurden, besteht das Terrain aus Satellitendaten, die von einem zusätzlich entwickelten Programm in ein Flächenmodell konvertiert wurden. Aufgrund des großen Gebietes wurden die Daten in Kacheln unterteilt, die vom System bei Bedarf nachgeladen werden können.

Unsere Systemarchitektur liefert im Falle der Anwendung „VR Mittelrheintal“ sehr zufrieden stellende Ergebnisse, so dass problemlos zahlreiche Modelle gleichzeitig im Speicher gehalten werden können, ohne die echtzeitfähige Darstellungsgeschwindigkeit zu verringern. Die Applikation wird in unserem Fall in Stereo dargestellt, wofür zwei Dual Xeon PCs (3 GHz) und ein AMD Athlon 3000+ zur Steuerung und Simulation eingesetzt werden. Alle Rechner sind mit 1 GB Arbeitsspeicher ausgestattet und über ein Gigabit-Ethernet unmittelbar verbunden. Die Navigation erfolgt über ein handelsübliches, kabelloses Gamepad mit zwei analogen Joysticks, das sich durch seine intuitive Bedienung für eine große Benutzergruppe als geeignetes Eingabegerät erwiesen hat.

Das komplette Storyboard mit allen Modellen, dem Szenenaufbau und den Handlungen wird in zwei Konfigurationsdateien definiert, die vom StateChartInterpreter verarbeitet werden. Die darin referenzierten Modelle und optionalen Kollisionsvarianten liegen im VRML-Format vor; eine Unterstützung von Modellen unterschiedlicher Auflösung (level of detail) ist ebenfalls möglich. Insgesamt gab es trotz der unkomplizierten Herangehensweise zur Verarbeitung der Ereignisse (Integration in die Rendschleife) keinerlei Performanzengpässe, was ein Indiz für das effiziente Zusammenspiel der einzelnen Komponenten des Systems selbst bei den großen Datenmengen wie im Falle unserer VR-Umgebung „Mittelrheintal“ ist.

## **7 Danksagung**

Wir möchten uns im Rahmen des Projektes bei allen Beteiligten bedanken. Insbesondere gilt unser Dank der DLR-DFD OE Umwelt- und Geoinformation in Weßling für die Bereitstellung der Terraindaten. Ebenso wäre die detaillierte Nachmodellierung der Marksburg nicht ohne die freundliche Unterstützung des Deutschen Burgenvereins e.V. unter der Leitung von Herrn Wagner denkbar gewesen. Die Festung Ehrenbreitstein wurde auf der Grundlage von Katasterplänen des Landesmuseums Rheinland-Pfalz modelliert, für deren Bereitstellung wir uns ebenfalls bedanken möchten.

## **8 Literatur**

- [Eb1993] Ebert, J.: Efficient Interpretation of State Charts. In (Esik, Z.,Hrsg.): Fundamentals of Computation Theory (FCT '93, Szeged, Ungarn), Nr. 710, S. 212-221, Springer-Verlag, 1993.

[IEC14772-1] International Standard ISO/IEC 14772-1:1997: The virtual reality modeling language, 1997, <http://tecfa.unige.ch/guides/vrml/vrml97/spec/>

[OSG2003] The OpenSG Forum: OpenSG. 2003, <http://www.opensg.org>

[RuJaBo1999] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language reference manual: The definitive reference to the UML from the original designers. 1. Auflage, Addison-Wesley, Reading, Mass., 1999

[W3C1996] The World Wide Web Consortium: Extensible Markup Language (XML). 1996, <http://www.w3.org/XML>



Abbildung 5: Der Bergfried der Marksburg vom Torgang der unteren Batterie



Abbildung 6: Das Deutsche Eck mit Blick auf das Kaiserdenkmal



Abbildung 7: Blick von der Kurtine auf das Ravelin (Festung Ehrenbreitstein)



Abbildung 8: Die Kemeate im gotischen Saalbau der Marksburg