

OpenLaszlo

Barrierefreie DHTML- und Flash-Transformation im Fokus*

Timo Ernst[†]
Universität Ulm[‡]

2. Juni 2008

*Version 1.2

[†]<http://www.timo-ernst.net>

[‡]Seminar Internetdienste: Rich Internet Applications (*Prof. Dr. Franz Schweiggert, Norbert Heidenbluth*)

Inhaltsverzeichnis

1	Einführung	3
1.1	Das Web 2.0	3
1.2	Definition	4
1.2.1	Eigenschaften von RIA's	4
1.2.2	Bekannte Beispiele[1] für RIA's	4
2	OpenLaszlo	5
2.1	Definition und Motivation . . .	5
2.2	Die Technik	5
2.2.1	Installation	5
2.2.2	LZX	5
2.2.3	Der Server	7
2.2.4	Der Interface-Compiler .	9
2.2.5	Beispiel einer einfachen Transformation	12
2.2.6	Client-Struktur	12
2.3	Security Model	13
2.4	Warum Flash?	13
3	OpenLaszlo und Barrierefreiheit	15
3.1	Definition	15
3.2	Das Dilemma von OpenLaszlo .	15
4	Quellen	16

Zusammenfassung

Dieses Paper behandelt das Open Source Projekt «OpenLaszlo» und betrachtet die Motivation und Funktionsweise hinter dieser Technologie.

Ein besonderer Fokus liegt auf der Konvertierung in Adobes Flash Format und DHTML, wobei weniger auf die Syntax der OpenLaszlo-Sprache LZX eingegangen wird, sondern mehr auf die eigentliche Technologie und die Idee dahinter.

Weiterhin wird auf barrierefreie WWW-Ressourcen eingegangen und OpenLaszlo auf dieses Thema hin untersucht.

1 Einführung

The phrase «Rich Internet Application Framework» is a lot like the word pornography: Easy to identify but hard to define [1]

Mit diesen Worten wird auf *java.net* ein kurzes Tutorial zu OpenLaszlo gegeben. William Grosso, der Autor, vergleicht die tägliche Arbeit eines Java-Programmierers mit dem Film «Und täglich grüßt das Murmeltier» mit Bill Murray, in dem die Hauptfigur immer wieder den gleichen Tag erlebt, ohne dass es dafür einen Grund zu geben scheint. Jeder Tag unterscheidet sich ein wenig von den anderen. Wenn er ein Erlebnis hatte, lernt er etwas daraus. Am Ende des Films schafft es Bill Murray aus dieser Schleife zu entfliehen und sein Leben weiterzuleben. Betrachtet man die Entwicklung von Webtechnologien wie PHP, JavaScript, Flash usw., so scheint alles ein kleines Stückchen besser zu werden, wie im Film, aber gleichzeitig gibt es immer wieder neue Frameworks, die das gleiche Problem erneut lösen wollen. Grosso betont dabei, dass diese Problematik sich nicht nur auf das WWW beschränkt sondern auch bei den klassischen

Desktop-Applikationen gilt. Als Beispiel nennt er die Packages swing und awt in JAVA, die beide für die Realisierung von Graphical User Interfaces gedacht sind, sich jedoch in einigen Details unterscheiden. Ein Paket, das die Vorteile von beiden Welten vereinen würde, wäre sicherlich sinnvoller.

Um nun zumindest im WWW diesem Kreislauf zu entrinnen gibt es eine ganze Reihe Ansätze. Einer davon sind die sog. RIA's (Rich Internet Applications). Diese werden spätestens seit Beginn der Ära des Web 2.0 immer häufiger eingesetzt und sollen langfristig gesehen zum Standard für zukünftige Webapplikationen werden.

1.1 Das Web 2.0

Das Web 2.0, oder auch das Social Web genannt, revolutionierte das WWW Anfang des neuen Jahrtausends zwar nicht (obwohl man das angesichts des Begriffs vermuten könnte), jedoch brach eine, nicht zuletzt durch die Medien ins Rollen gebrachte, Lawine los die bis heute nicht zum Erliegen gekommen ist.

Überall schießen Web 2.0-Dienste aus dem Boden wie Pilze und der unendliche Sumpf des «Netzes des Netze» wird überflutet von Ideen von denen man oft das Gefühl nicht los wird, alles wäre schon einmal dagewesen. Nur ohne 2.0.

Doch schaut man einmal hinter die Kulissen, wird deutlich, dass der Hype um das Web 2.0 nicht nur eine Fülle von Inhalten sondern auch viele neue Technologien hervorgebracht hat. So zum Beispiel RIA's. Sie sollen statische HTML-Seiten lieber heute als morgen ablösen. Ziel ist es, Webseiten so erschein- und bedienbar zu machen, als würde man eine lokale Desktop-Applikation bedienen.

Eine für diesen Zweck entwickelte Technologie ist OpenLaszlo, mit der es möglich ist, über die selbstentwickelte Sprache LZX, RIA's zu erstellen, die auf heute üblichen Standards

aufsetzen (HTML, JavaScript, Flash).

Ziel ist es, Layouts zu vereinheitlichen während das Äußere an, aus gängigen Betriebssystemen gewohnte, Fenstermanager-Stile angepasst werden soll. Dem Entwickler wird dabei ein Teil der Arbeit der Designentwicklung abgenommen und der eigentliche Inhalt stärker in den Vordergrund gerückt.

1.2 Definition

Wie zu Beginn dieses Papers erwähnt, ist es sehr schwierig RIA's zu definieren. Allerdings können einige Eigenschaften oft beobachtet werden:

1.2.1 Eigenschaften von RIA's

Rich Internet Applications...

... werden oft aus einer Webseite heraus oder innerhalb dieser ausgeführt.

... geben dem User sofortiges Feedback ohne Delay, der bei Webanwendungen normalerweise vorkommt.

... verwenden moderne User Interface Controls, wie zum Beispiel Baumdarstellungen und Karteikarten anstatt auf den künstlerisch-geistigen Erguss von Webdesignern angewiesen zu sein.

... erlauben dem Benutzer, sog. «Fat Client Operations» zu nutzen. Diese sind z.B. Drag 'n Drop oder Keyboard-Navigations-Semantiken, wie man sie aus Fenstermanagern von Betriebssystemen wie Windows oder Linux kennt.

... implementieren alle oben genannte Punkte mit Hilfe von platform- und browserunabhängigen Technologien, wie zum Beispiel JavaScript.

... nutzen alle ein clientseitiges Containermodell.

... laden nicht ganze HTML Seiten, sondern behandeln nur die lokalen Klicks, ohne den Server extra noch einmal anzufordern und erleben durch die Minimierung der Serveranfragen einen Performanceboost.

... nutzen die Tatsache aus, dass Webbrowser nahezu überall vorhanden sind. Daraus folgt ein oft massiver Einsatz von JavaScript und/oder Flash

... nutzen oft XML bzw. einen XML-Dialekt, der mit Hilfe eines Präprozessors in DHTML konvertiert wird.

1.2.2 Bekannte Beispiele[1] für RIA's

- XAML (Microsoft)
- Asperon (Java Technologie)
- Java Web Start
- General Interface (baut auf einer JavaScript Library auf)
- Ideaburst (Auf SVG-Basis)
- Kenamea (Ersetzt HTTP durch ein eigenes Protokoll und nutzt JavaScript + DHTML auf Clientseite)
- XUL (Ein graphisches Toolkit von Mozilla)
- Laszlo, Snapp MX, Flex (Auf Flashbasis)

2 OpenLaszlo

2.1 Definition und Motivation

OpenLaszlo ist ein Framework auf Flash- und DHTML-Basis, das eine Entwicklungsumgebung zur Erstellung von RIA's bietet und wurde am 07. Oktober 2004 von Laszlo Systems unter dem Namen «Laszlo Presentation Server» entwickelt. Diese Plattform, welche heute ein Open Source Projekt ist, war damals noch proprietär. Die Idee war, eine RIA auf Flashbasis «on the fly» entwickeln zu können um diese später an große Firmen zu verkaufen. Heute läuft das Projekt unter der Common Public License (CPL) und ist frei verfügbar.

2.2 Die Technik

Die Technologie ist deklarativ serverbasierend und besteht aus dem XML-Dialekt LZX und dem Application Server (auch «Presentation Server» genannt), der zugleich Compiler für LZX Source Code ist.

2.2.1 Installation

Da der bereits erwähnte Generator auf JAVA basiert, ist die JAVA Entwicklungsumgebung JDK (Java Development Kit) und **nicht nur** die JAVA Laufzeitumgebung JRE (Java Runtime Environment) auf dem System erforderlich. Beides kann kostenlos von der SUN-Webseite <http://java.sun.com> heruntergeladen werden (MacOS-Benutzer benötigen das JDK nicht, da es bereits im Betriebssystem enthalten ist, allerdings ist ein Update auf die neueste Version ratsam).

Neben dem JDK muss anschließend nur noch der OpenLaszlo Server selber installiert werden (Der für die Kompilierung erforderliche Tomcat Server ist hierbei bereits enthalten). Diesen erhält man von <http://www.openlaszlo.org/download> und ist für Windows, MacOS und Linux erhältlich. Zur

Installation unter Windows muss lediglich das setup-file ausgeführt werden.

Nach der Installation sollte im Startmenü eine neue Gruppe mit der Bezeichnung «OpenLaszlo Server» existieren (Start, Programme, OpenLaszlo-Server). Hier wählt man einfach «Start OpenLaszlo Server» und das System sollte anlaufen. Erkennen kann man dies am Log-Fenster vom Tomcat. Verschwindet dieser, ist dies ein Zeichen, dass der Tomcat-Server terminiert ist. Es ist unbedingt darauf zu achten, dass keine anderen Dienste auf Port 8080 hören, denn dieser sollte für den Tomcat-Server reserviert sein.

Um zu testen, ob die Installation erfolgreich war, kann man nun im Browser die mitgelieferte *hello.lzx* aufrufen. Wird die «Hello Laszlo» Meldung nicht angezeigt, ist etwas schief gegangen.

Die offizielle Dokumentation zu OpenLaszlo ist dabei immer eine gute Hilfe: <http://www.openlaszlo.org/documentation>

Läuft der Server, kann mit der Entwicklung erster OpenLaszlo-Anwendungen begonnen werden. Handwerkszeug hierfür ist die XML-Anwendung LZX.

2.2.2 LZX

Die Sprache LZX ist eine Auszeichnungssprache (Markup-Language) und definiert sowohl Seiten- als auch Userinterface-Konstrukte. Dabei kommt JavaScript zum Einsatz, um Prozeduren und Variablen zu verwalten. Dies hat zwei entscheidende Vorteile:

1. **Übersichtlichkeit**

Aufgrund der Ähnlichkeit zu XML bzw. HTML besteht ein hohes Maß an Lesbarkeit des Sourcecodes.

2. **Kompatibilität**

Da OpenLaszlo auf dem sehr verbreitetem

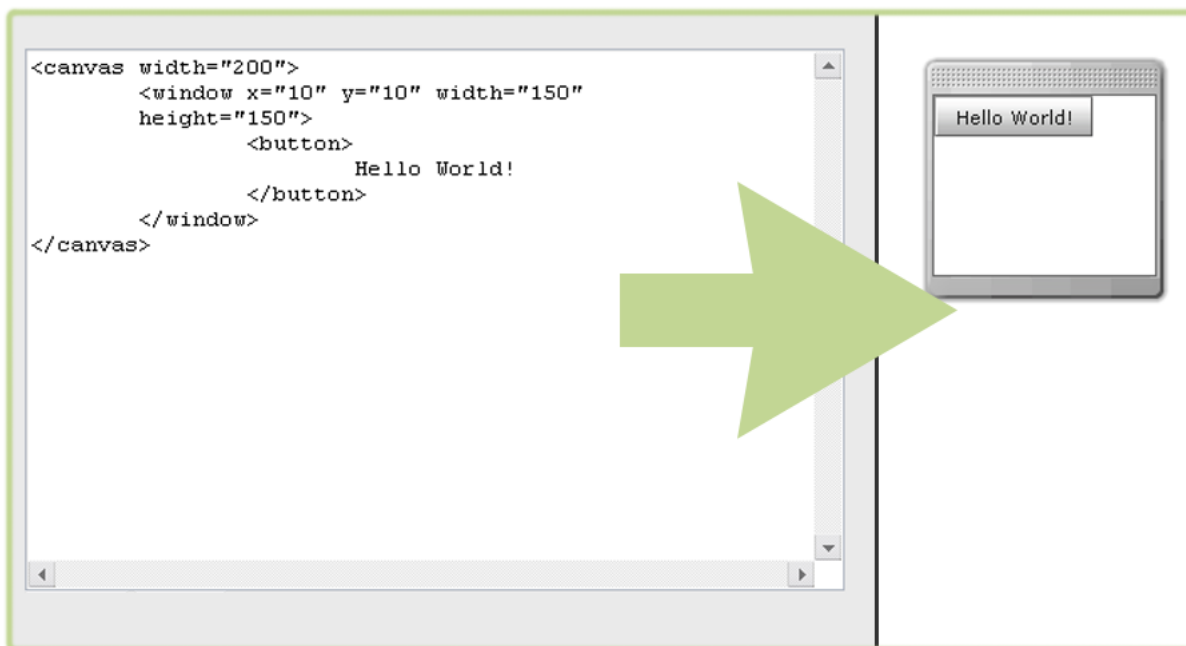


Abbildung 1: Ergebnis des Codes aus Beispiel B

JavaScript basiert, gibt es bezüglich der Kompatibilität kaum Probleme.

Laszlo-Universum ebenfalls mindestens einen Layoutmanager, genannt «ReverseLayout».

Beispiel A:

```
<button text="Login" onclick="login()"/>
<script>
  login_dialog.open();
  function login() {
    // irgendeine Funktion
  }
</script>
```

Scripte, wie in diesem Beispiel werden automatisch vom Server eingebunden. Allerdings ist bis zur Laufzeit unbekannt, ob das referenzierte Script bzw. die Funktion überhaupt valide ist, geschweige denn existiert.

Der Aufbau von LZX-Code ist immer gleich. Als Root-Umgebung dient `<canvas>...</canvas>`. Die Tag-Zeilen werden von oben nach unten abgearbeitet. Es gibt keine `main()`-Funktion, wie man sie aus anderen Sprachen her kennt, wie zum Beispiel JAVA. Eine Unähnlichkeit zu SUN's Objektorientierter Sprache lässt sich jedoch nicht leugnen. So zum Beispiel gibt es im

Beispiel B:

```
<canvas width="200">
  <window x="10" y="10" width="150" height="150">
    <button>
      Hello World!
    </button>
  </window>
</canvas>
```

Das Resultat aus Beispiel B sieht man in Abbildung 1. Bei diesem generierten Fenster ist es möglich, es zu verschieben und mit einem weiteren Parameter sogar skalierbar zu machen. Ergänzt man das Window nun um ein paar Eingabefelder (Wie in einem HTML-Formular) ist es mit Hilfe des Tabulators möglich von einem Feld zum Nächsten zu springen. Dabei wird eine kleine Animation angezeigt, die dem User ein Feedback darüber geben soll, was gerade passiert. Dies ist ein wichtiger Bestandteil von OpenLaszlo, auf welchen großer Wert gelegt wurde bei der Entwicklung. Laszlo Systems nennt dieses

Feature «Cinematic User Experience».

Natürlich müssen wichtige Daten auch irgendwo gespeichert werden. Theoretisch könnte man dies in einer SQL-Datenbank erledigen oder einfach JavaScript hierfür nutzen. Beide Ideen haben jedoch Nachteile: SQL-Datenbanken müssen auf dem Server installiert sein, während JavaScript-Variablen nicht persistent sind. Daher werden einfach XML Datasets gespeichert. Diese Technik ist übersichtlich und standardisiert. Ausgelesen wird, wie in XML üblich, mit XPath, wobei nicht das ganze Spektrum dieser Notation für URI's zum Einsatz kommt, sondern lediglich ein (nicht kleiner) Subset davon.

Der Aufbau einer XPath-URI funktioniert dabei folgendermaßen:

:	Dataset
[]	Auswahl eines Knotens
@	Attribut
text()	Liest einen Knoten aus

Beispiel:

```
<dataset name="students">
  <students>
    <person>
      <name>Timo Ernst</name>
      <matrikelnr>4580453425</matrikelnr>
    </person>
    <person>
      <name>Captain äBlaubr</name>
      <matrikelnr>3453452425</matrikelnr>
    </person>
  </students>
</dataset>
```

Eine mögliche URI hierbei wäre:
students:students[0]/name/text()

Ausgabe: *Timo Ernst*

2.2.3 Der Server

2002 wurde der erste OpenLaszlo Server aufgesetzt und 2004 offiziell *.swf4 (Shock-Wave-Format) und DHTML als Zieltyp aufgenommen. Um allerdings die Browserkompatibilität weiter voranzutreiben, sollen auch in Zukunft

weitere Formate unterstützt werden, wie zum Beispiel: swf9, SVG, JavaME oder Silverlight.

Der OpenLaszlo-Server hat die Aufgabe den Code zu «kompillieren». Dieser Begriff der Kompilierung ist allerdings nur dann zutreffend, wenn das Ausgabeformat ein binäres Flash-File ist. Ist stattdessen DHTML gewünscht, so wäre der Ausdruck «Transformation» eher zutreffend. Eine solche sollte theoretisch bei jedem Aufruf durch einen Browser stattfinden. Um jedoch Redundanzen zu minimieren hält der Server einen Cache und «kompiliert» den Code nur, falls sich dieser ändern sollte.

Die Konvertierung in das binäre Flashformat geschieht durch einen auf JAVA basierten Generator, welcher in einem J2EE Servlet Container ausgeführt wird, der in der Regel auf einem Apache Tomcat aufsetzt.

Beim Ausliefern der Applikation kennt OpenLaszlo zwei grundsätzlich unterschiedliche Herangehensweisen:

1. SOLO

Dies ist die übliche Herangehensweise: Der Entwickler lässt sich, lokal vom OpenLaszlo-Server, eine swf-Datei (Shockwave Flash Format) erstellen. Diese stellt er auf einem Webserver entweder direkt oder eingebettet in einer HTML-Seite bereit. Der Client kann diese Applikation nun über einen Webbrowser mit dem Flashplugin abrufen.

2. Über einen Proxy

Natürlich könnte die Auslieferung auch über einen Proxy geschehen. Dies hätte den Vorteil, dass man die Pakete, die übertragen werden, manipulieren könnte um zum Beispiel Werbung einzublenden, was bei HTML-Seiten die über HTTP-Proxys übertragen werden, oft der Fall ist.

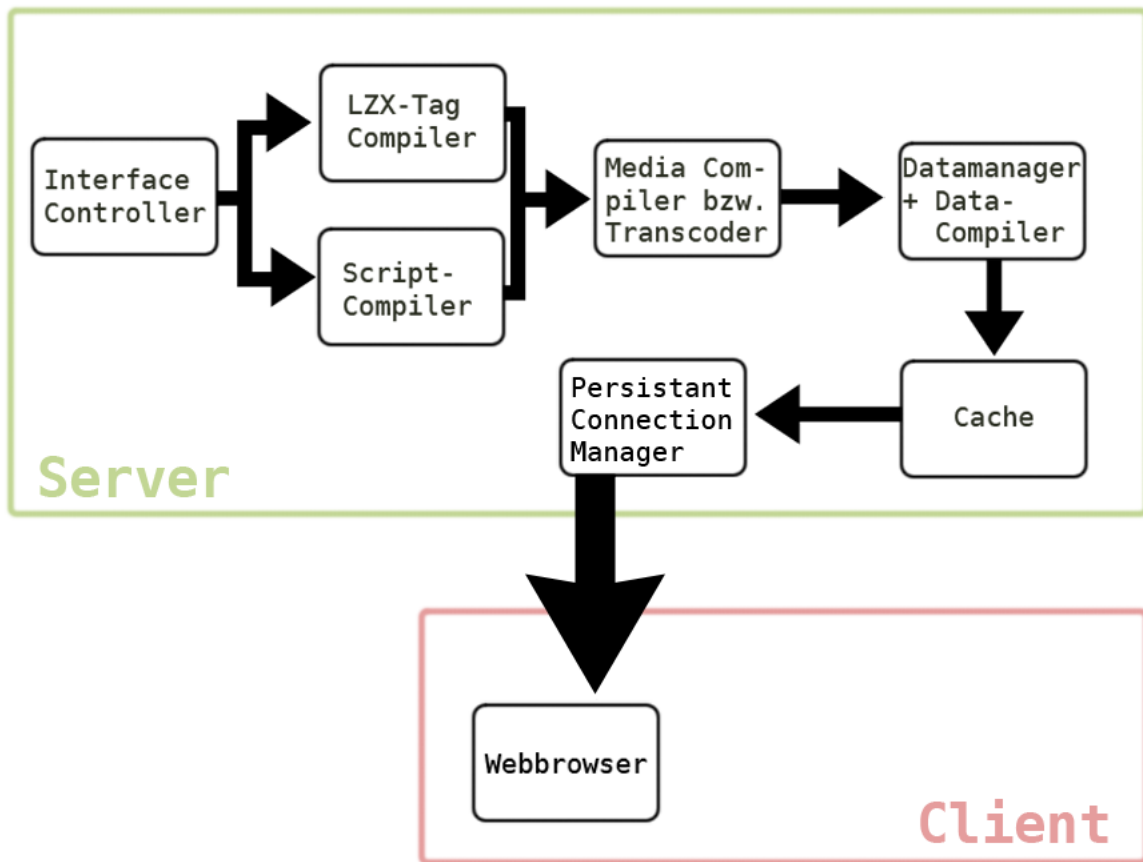


Abbildung 2: Ablauf der LZX-Transformation

Eine eher seltene Variante, dass sowohl Server als auch Client auf derselben Maschine laufen ist zwar denkbar, aber eher unüblich. Lediglich bei der Entwicklung von OpenLaszlo-Applikationen spielt dieses Setup eine Rolle.

Der übliche Ablauf sieht also so aus, dass der Client eine HTML-Seite aufruft, in der das Flashobjekt über das `<object>`-Tag eingebunden und anschließend über HTTP übertragen wird.

Natürlich kann auch der Client Datensätze an den Server schicken, indem er jedoch kein Binary, sondern ein XML-File versendet. Intern kommen bekannte Verfahren, wie zum Beispiel

das MVC-Pattern (Model View Controller) zum Einsatz, welches für eine Trennung zwischen Design und Programmlogik sorgen soll.

Der Server besteht dabei aus den folgenden Subsystemen (Abb. 2):

- Interface Controller/-Compiler
- Media Transcoder
- Data Manager
- Persistent Connection Manager
- Cache

Das erste Element, der *Interface Controller* (Auch *Interface Compiler* genannt) besteht dabei wiederum aus zwei Komponenten: Dem *LZX Tag Compiler* und dem *Script-Compiler*, deren Aufgabe es ist, LZX-Tags und JavaScript Code in swf-Dateien (Shockwave Flash Format) zu kompilieren (Natürlich nur dann, wenn Flash als Zielformat gewünscht wird, und nicht DHTML). Dies geschieht mit Hilfe des *Media Compilers* und des *Data Managers* (beinhaltet einen *Data Compiler*, der alle Daten in ein komprimiertes Binärformat konvertiert), welche aufgerufen werden um mediale (zum Beispiel Bilder) und sonstige Daten in das Zielformat zu kompilieren. Erstere Konvertierung wird mit Hilfe des *Media Transcoders* realisiert, dessen Aufgabe es ist, diverse Medientypen umzuwandeln (z.B. JPEG, GIF, PNG, MP3, usw). Dieser Code wird dann in den Cache geladen und von dort mit Hilfe des *Persistent Connection Managers* zum Client geschickt. Dieser kümmert sich um die Verbindung zwischen Client und Server, wobei die Betonung auf «Persistent» liegt. Dies bedeutet, dass, während die Applikation läuft, immer eine Verbindung zum Server vorhanden ist, obwohl die Anwendung augenscheinlich «nichts macht».

Damit löst sich OpenLaszlo erfrischend von der klassischen Methode, für jede Ressource (sei es ein Bild oder ein HTML-Dokument) eine eigenen HTTP-Request abzuschicken.

2.2.4 Der Interface-Compiler

Obwohl die gesamte OpenLaszlo Architektur sehr gut im WWW dokumentiert ist, ist es sehr schwierig herauszufinden, was denn der Compiler intern nun eigentlich genauer macht. Aus diesem Grund wandte ich mich für dieses Paper direkt an die OpenLaszlo Entwickler, wo mir Henry Minsky, Software Architekt bei Laszlo Systems, freundlicherweise einen kleinen Überblick über die Thematik gab, indem er mir den

groben Ablauf eines Compilervorgangs erläuterte.

Gestartet wird ein Kompilierungsvorgang, indem der Server den Compiler über den Compilation Manager initialisiert, wenn er die Aufforderung zur Kompilierung einer Anwendung bekommt, die NICHT im Cache liegt. Das Build-System nutzt das LZX-Kommandozeilentool, um die Applikation zu testen («Smoketesting»).

Der Compiler besteht dabei aus:

- Parser
- Schema
- Validator
- Script-Compiler
- Element Compiler

Letztgenannter Element Compiler untersucht rekursiv die Elemente innerhalb der Sourcefiles und ihrer, über Include-Anweisungen eingebundenen, Dateien und schreibt den daraus resultierenden Bytecode in das Outputfile. Anschließend wird der Script-Compiler gerufen, um ECMAScript (=JavaScript) in Object-Byte zu kompilieren. Im dritten Schritt müssen externe Medientypen kompiliert werden. Dazu zählen:

1. Inline Bilder
2. Videos
3. Audio Files
4. XML Datensätze

Diese Aufgabe wird, wie bereits erwähnt, vom Media- und Data Compiler übernommen, welche übrigens auch genutzt werden, während die Applikation bereits läuft, um die Anzahl Requests an den Server zu minimieren.

Der eigentliche Kompilervorgang kann dabei in die folgenden vier Schritte aufgeteilt werden:

1. Parsing

Als Parsing bezeichnet man, *im OpenLaszlo-Kontext*, den Prozess der Konstruierung einer erweiterten DOM Repräsentation einer Ressource.

Diese besteht aus 6 Schritten:

(a) Dereferenzierung

Der Name der zu parsenden Datei wird auf eine Ressource im lokalen Dateisystem dereferenziert.

(b) Parsing

Ein SAX-Parser liest den Input-Stream und schickt diesen an das...

(c) Pre-Transform DOM (=Document Object Model)

Dieser benutzt einen Content-Handler, um SAX Events und den Pfad zur Quelle abzufangen. Anschließend wird ein DOM aus den Custom-Elementen gebaut, der die Quellpfade als Felder einbindet.

(d) Preprocessing

Hierbei wird SAXON eingesetzt, um ein XSLT Stylesheet-Dokument (`lps/schema/preprocess.xsl`) zu erstellen. Dieses aktualisiert den Namensraum und fügt einen solchen zu Elementen hinzu, die keinen haben.

(e) Post-Transform DOM

Hier wird ein JDOM Model aus den Events, die vom Präprozessor resultieren gebaut. Es wird eine selbst geschriebene Klasse benutzt, die die Quellpfad-Attribute in die Felder der Elemente parst, die Subklassen der JDOM Elemente sind.

(f) Expand

Expand ersetzt include-Statements, die nicht aus der OL Library stammen, durch ihre Ziele, die rekursiv geparst werden. Ein Stack, bestehend aus den zur Zeit bearbeiteten Ressourcen wird genutzt, zum rekursive Includes zu suchen und zu finden.

2. Schema Update

Hierunter versteht man das Kopieren und Aktualisieren des Default Schemas nach den Klaskendefinitionen in der main DOM und den Library-Includes.

3. Validierung

Die Ziele der main-DOM und Library-Includes werden mit dem aktualisieren Schema verglichen und validiert. Validation Warnings und Errors werden in die Liste der Compiler Warnings gesammelt. Interessant ist an dieser Stelle noch zu erwähnen, dass ein Fatal Validation Error einer Non-Fatal Compiler Warning entspricht.

4. Script-Compiler

Wie der Name schon andeutet, führt der Script-Compiler eine Umwandlung von ECMAScript in ActionScript-Bytecode aus. Dabei werden die folgenden Stadien durchlaufen:

(a) Parsing

Der Sourcecode wird zunächst mit Hilfe von JavaCC (=«Java Compiler Compiler», ein Parser-Generator, der JAVA-Code produziert) untersucht und anschließend in einen Parse-Tree umgewandelt.

(b) Code Generation

Der Parse-Tree wird in eine Folge von Objekten umgewandelt, die ActionScript-Instruktionen repräsentieren.

(c) **Assemblierung**

Die in Schritt 4b erstellte Instruktionsequenz wird in eine Bytefolge kompiliert.

Während der Parser in JAVA geschrieben ist, wurde für den Compiler Python-Code benutzt, welches mit Hilfe von Jython in JVM Bytecode umgewandelt wird und somit auf jeder JAVA-fähigen Maschine lauffähig ist.

5. Element Kompilierung

Unter diesem Begriff versteht man die Untersuchung der XML-Elemente des Dokuments. Dieses Wissen wird genutzt, um Informationen über das Object File zu sammeln. In den Sourcefiles wird dieser Schritt auch oft einfach nur «Compilation» genannt.

Während der Bearbeitung werden die XML-Elemente eines Dokuments rekursiv bearbeitet und die Information in ihnen benutzt, um eben solche in einem Object-File zu sammeln. Der Compiler ist als ein Satz von Klassen implementiert, die *ElementCompiler* erweitern. Verschiedene Subklassen kompilieren unterschiedliche Arten von Elementen (Script-Tags, Font Tags, View, andere Node Tags...). Jede Subklasse enthält Methoden, die das Schema mit Tags, die das Element und die Ausgabe-Objekte definiert, abhängig vom Inhalt des Elements, aktualisieren.

Der Compiler erstellt eine Instanz von *CanvasCompiler* und fügt diese an die Dokumentwurzel an. *CanvasCompiler* wiederum erweitert *TopLevelCompiler*, der Instanzen von *ElementCompiler* zu jedem der Kinder eines Knotens hinzufügt.

Die Compiler-Methode eines *ElementCompilers* kann eine der folgenden

Operationen ausführen:

- Compiler auf dem Kind des Elements aufrufen (*CanvasCompiler*, *LibraryCompiler*. Erweitern beide *TopLevelCompiler*).
- Asserts zu einer Object-Datei hinzufügen (*ResourceCompiler* und *FontCompiler*)
- ECMAScript generieren, welches in Bytecode kompiliert wird.
- Bytecode direkt in das Object-File hinzufügen.

Die genaue Implementierung des Compilers findet sich im Package *org.openlaszlo.compiler* und die des Script-Compilers in *org.openlaszlo.sc*.

Um diese ganzen Einzelschritte etwas kompakter darzustellen, kann der Kompilierungsvorgang im Prinzip in zwei große Schritte unterteilt werden:

1. Die erste Phase bearbeitet alle `<include>`-Tags, um den gesamten Sourcecode zu sammeln, ähnlich dem Präprozessor in C++. Anschließend wird nach allen Klassendeklarationen gesucht, um ein Modell der Klassenhierarchie im Speicher aufzubauen. Als letztes wird noch gespeichert, von welchem Typ alle deklarierten Attribute sind.
2. Die zweite Phase prüft alle Tags des Codes, die Instanzen einer Klasse repräsentieren und entsprechend instantiiert werden müssen. Das sind Elemente vom Typ: `<view>`, `<node>` und alle Subtypen davon.

Nachdem beide Phasen beendet wurden, werden alle JavaScript Statements geschrieben, die an die Runtime übergeben werden.

Dabei werden alle Werte von Attributen, die Constraints deklarieren, so annotiert, dass sie der Script-Compiler findet und entsprechende Constraint-Abhängigkeiten hinzufügt.

Der JavaScriptcode, den der Tag-Compiler produziert ist i.d.R. eine (tief verkettete) Liste, die alle Informationen hält, um Klassen und Views mit all ihren eingebetteten parent-/child Beziehungen zu instantiieren.

Beim Starten der Anwendung verrichtet die LFC (=«Laszlo Foundation Class») übrigens viel Arbeit um diese JavaScript-Listen in eine View- und Classhierarchie zusammenzusetzen. Minsky lies bei seinen Erklärungen durchblicken, dass für die Zukunft mit SWF9 geplant ist, dass der Tag-Compiler mehr Arbeit verrichten soll, um zum Beispiel Initialisierungsfunktionen deklarieren zu lassen. Dies soll innerhalb von echten ECMA4-Style Deklarierungen realisiert werden, die effizienter kompiliert werden können, was zu einer höheren Performance beiträgt und die LFC am Clienten etwas Arbeit abnehmen soll.

2.2.5 Beispiel einer einfachen Transformation

Transformation von:

```
<view name="foo">
  <view width="50" height="50" bgcolor="red"/>
  <text y="50">This is some text</text>
</view>
```

in diesen JavaScriptcode:

```
LzInstantiateView({attrs: {name: "foo"},
  children: [{attrs: {bgcolor: 0xff0000
    ,height: 50
    ,width: 50
  }, name: "view"}, {attrs: {
    font: "Verdana,Vera,sans-serif"
    ,fontsize: 11
    ,fontstyle: "plain"
    ,text: "This is some text", y: 50
  }, name: "text"}], name: "view"},
  3);
```

Hierbei handelt es sich um eine eingebettete Liste, die eine Child-View Hierarchie darstellt,

mit allen dazugehörigen Attributen. Im Falle von Methoden und Eventhandlern werden diese auch als Attributwerte übergeben.

Intern werden noch weitere Transformationen durch den Script-Compiler auf dieses Stück JavaScript ausgeführt, um z.B. Constraint Funktionen zu behandeln oder JavaScript 2.0 Klassendeklarationen in JavaScript 1.0 kompatiblen Code zu konvertieren.

Abschließend gab Minsky noch den Tipp über die Kommandozeile den Compiler wie folgt aufzurufen:

```
lzc -script foo.lzx
```

Dieser -script-Parameter gibt den JavaScript-Code aus, der dem Script-Compiler übergeben wird.

2.2.6 Client-Struktur

Dem Client wird bei der Übermittlung nicht nur der eigentliche Inhalt der Applikation übermittelt, sondern auch die sog. Corelibrary. Diese wird immer in jede OpenLaszlo Anwendung hineinkompiliert, die run-Time Services (z.B. einen Timer) und/oder einen Presentation-Renderer benötigen, um zum Beispiel 2D-Grafiken darzustellen und Sound abzuspielen. Diese Corelibrary wird «Laszlo Foundation Class» (LFC) genannt.

Flash ist hierbei nicht zwingend notwendig. Wenn eine Kompilierung in das SWF-Format geschieht, wird der Flashplayer lediglich als Rendering Engine genutzt.

Bestandteile der Client-Architektur:

1. Event System

OpenLaszlo nutzt das Konzept von Events, die benötigt werden um Mausklicks oder einen Datapush vom Server zu erkennen. Diese Ereignisse

werden vom Event System erkannt und lokal bearbeitet, was zur Folge hat, dass die Last am Server reduziert wird.

2. Data Loader/Binder

Dieses System dient als Koordinator des Datentraffics. Es akzeptiert Datenströme vom Server und bindet diese an visuelle Elemente, wie zum Beispiel Textfelder oder Forms. Dieses Verhalten ist für eine Applikation aus dem WWW eher untypisch und daher bemerkenswert, da im «klassischen World Wide Web» das, von Tim Berners Lee entwickelte, HTTP Protokoll ein klassisches Request-Reply Verfahren nutzt und nicht vorsieht, dass der Server von selbst Daten an den Client schickt.

3. Layout- und Animationssystem

Dieser Teil der LFC stellt das Screen Layout von Interface-Elementen. Dabei kann bei der Positionierung der Interface-Elemente zwischen relativer und absoluter Positionierung gewählt werden. Weiterhin sind in diesem System Algorithmen implementiert, um zum Beispiel eine Animation eines Fensters zu realisieren, wenn dieses geschlossen wird. Gerade diese Animationen sind ein wichtiger Bestandteil der Philosophie von OpenLaszlo, die dem Benutzer Statusänderungen so gut wie möglich visuell suggerieren sollen. Dies erlaubt das Erstellen von dynamischen Applikationen mit wenig Code.

2.3 Security Model

Um bei der Übertragung von sensitiven Daten eine hohe Sicherheit zu gewährleisten unterstützt OpenLaszlo auch SSL über HTTPS. Weiterhin gibt es, ähnlich wie bei JAVA Applets, das Konzept des lokalen Sandboxmodells, in dem SWF-Dateien lokal ausgeführt werden. Innerhalb dieser Sandbox können die

Anwendungen nicht ins Dateisystem schreiben oder die native Umgebung des Browsers manipulieren. Bestimmte Webdienste und Datenbanken, die von der OL-Applikation genutzt werden, werden durch ein «Per-User Authentication Model» geschützt. Dies führt zur Vermeidung des Missbrauchs des OL-Servers als Proxy zu ungeschützten Diensten/Daten.

2.4 Warum Flash?

Flash wurde ursprünglich als Autorensystem konzipiert um multimedialen Inhalt, wie Flash-Filme, einfach in's WWW zu bringen. Durch die konsequente Erweiterung dieser Technologie hat sich Flash nicht zuletzt dank Youtube und Co. durchgesetzt und ist seit dem eine sehr weit verbreitete Technologie, so dass inzwischen so gut wie jeder Browser, der täglich im Einsatz ist, Flash unterstützt. OpenLaszlo-Applikationen laufen auf jedem Flashplayer, der mindestens swf7-Dateien abspielen kann.

Zum Vergleich: 2007 waren u.a. Flash und weitere Webtechnologien wie folgt verbreitet:

1. ||| Flash: 98.8 %
2. ||| Java: 84.6 %
3. ||| WMF: 83 %
4. ||| Quicktime: 68,4 %
5. ||| RealPlayer: 52,6 %

Quelle: adobe.com (2007)

Einer der Gründe für diese Bekanntheit ist die Erfinderfirma Macromedia, die großen Aufwand betrieb, um ihre Technologie ubiquitär zu machen.

Der größte Vorteil von Flash gegenüber HTML ist sicherlich, dass das Verhalten und das Aussehen in jedem Browser identisch ist,

da dieser das Flash-Binary lediglich einbettet. Der Flash Player kümmert sich dann um den Rest. (D)HTML Webseiten dagegen können in jedem Browser unterschiedlich aussehen. Dies liegt an der Interpretation, die jeder Browser anders auslegt, obwohl es von Seiten des W3C (World Wide Web Consortium) strikte Vorgaben bezüglich der HTML Syntax gibt.

3 OpenLaszlo und Barrierefreiheit

3.1 Definition

Auszug aus dem deutschen Gesetz zur Gleichstellung behinderter Menschen (§4):

«Barrierefrei sind bauliche und sonstige Anlagen, Verkehrsmittel, technische Gebrauchsgegenstände, Systeme der Informationsverarbeitung, akustische und visuelle Informationsquellen und Kommunikationseinrichtungen sowie andere gestaltete Lebensbereiche, wenn sie für behinderte Menschen in der allgemein üblichen Weise, ohne besondere Erschwernis und grundsätzlich ohne fremde Hilfe zugänglich und nutzbar sind.»

Oft wird der Begriff der Barrierefreiheit auch durch das Wort «Behindertengerecht» ersetzt. Allerdings ist dies nicht ganz korrekt, da barrierefreie Bedienschnittstellen für *alle* Menschen zugänglich sein sollten. Da körperlich eingeschränkte Menschen jedoch weniger Möglichkeiten haben als diejenigen ohne Behinderung und die Definition der Barrierefreiheit aus dem Gesetz zur Gleichstellung behinderter Menschen stammt, verschwimmt die genaue Trennung zwischen diesen Begriffen oft.

Das W3C geht einen Schritt weiter und stellt im Zusammenhang mit dem Begriff der Webentwicklung die folgenden Richtlinien auf:

- Stellen Sie äquivalente Alternativen für Audio- und visuellen Inhalt bereit.
- Verlassen Sie sich nicht auf Farbe allein.
- Verwenden Sie Markup und Stylesheets und tun Sie dies auf korrekte Weise.
- Verdeutlichen Sie die Verwendung natürlicher Sprache.

- Erstellen Sie Tabellen, die geschmeidig transformieren.
- Sorgen Sie dafür, dass Seiten, die neue Technologien verwenden, geschmeidig transformieren.
- Sorgen Sie für eine Kontrolle des Benutzers über zeitgesteuerte Änderungen des Inhalts.
- Sorgen Sie für direkte Zugänglichkeit eingebetteter Benutzerschnittstellen.
- Wählen Sie ein geräteunabhängiges Design.
- Verwenden Sie Interim-Lösungen.
- Verwenden Sie W3C-Technologien und -Richtlinien.
- Stellen Sie Informationen zum Kontext und zur Orientierung bereit.
- Stellen Sie klare Navigationsmechanismen bereit.
- Sorgen Sie dafür, dass Dokumente klar und einfach gehalten sind.

Der Grund für diese Richtlinien sind oft sehbehinderte Menschen, die sog. Screenreader benutzen, um sich den Inhalt einer Seite vorlesen zu lassen. Dies sind Geräte oder Anwendungen, die auf dem Bildschirm dargestellten Text entweder akustisch oder über eine Braillezeile wiedergeben, die den angezeigten Text auf einer Oberfläche für den Anwender ertastbar macht. Ist der HTML-Code der Seite jedoch von diesem Reader nicht interpretierbar, so ist der Benutzer ausgesperrt.

3.2 Das Dilemma von OpenLaszlo

Das Problem hinsichtlich der Barrierefreiheit, bei mit OpenLaszlo erstellten Webseiten, ist die verwendete Technologie. Zwar pro-

duziert der Server beim Transformationsprozess sauberen HTML-Code, jedoch wird immer DTHML oder Flash benutzt und beide Technologien sind gerade von älteren Screenreadern nur schwer oder gar nicht interpretierbar.

Flash kann aufgrund seiner binären Architektur nicht oder nur selten ausgelesen werden, was auch den Bots von Suchmaschinen, wie Google und Co. das Leben schwer macht. Zwar gibt es seit Version 6 von Flash einige Verbesserungen hinsichtlich der Barrierefreiheit, die manchen Screenreadern das Interpretieren der Inhalte ermöglicht, jedoch sind diese noch nicht sehr verbreitet und nur von wenigen Geräten unterstützt. So zum Beispiel werden textliche Inhalte inzwischen in einer XML-Datei gespeichert, die problemlos von Screenreadern, Braillezeilen, Bots oder textbasierten Browsern gelesen werden kann.

Weiterhin ist es seit Flash MX möglich, Fehler genauer zu spezifizieren und dem Screenreader Hinweise bezüglich der Inhalte zu geben, ähnlich dem ALT-Attribut in HTML.

Ein weiteres Feature sind Kurzbefehle, die die Navigation vereinfachen sollen (z.B. Tastenkombinationen um bestimmte Aktionen durchzuführen bzw. Buttons zu drücken. Realisiert wird das durch die von Microsoft entwickelte MSAA-Schnittstelle, die als Brücke zwischen Webinhalt und Screenreader darstellt. Zwar unterstützt die aktuellste Version von Flash nun diese Schnittstelle, ältere Geräte jedoch können damit nichts anfangen. Um solche Probleme zu umgehen und für die Screenreader-Software keine eigene Engine entwickeln zu müssen, werden inzwischen verbreitete Webbrowser wie zum Beispiel Mozilla Firefox benutzt, und aus diesem mit Hilfe eines Plugins die Daten ausgelesen.

JavaScript-basierte Webseiten dagegen sind zwar im Plaintext lesbar, benötigen jedoch einen Interpreter, der diesen Quelltext sinnvoll umsetzt, was bei Screenreadern oft nicht

möglich ist. Dieses Problem tritt besonders bei Webseiten auf, die auf der zur Zeit sehr beliebten Technologie, AJAX basieren. Das Dilemma hierbei liegt darin, dass mit AJAX/JavaScript interaktive Webseiten erstellt werden können, die Bedienmetaphern wie Drag 'n Drop oder das Minimieren, Maximieren und Verschieben von Fenstern, die aus der Desktop-Welt bereits bekannt sind, unterstützen. Da OpenLaszlo gerade von letzterem Feature exzessiv Gebrauch macht ist eine Unterstützung für Sehbehinderte Menschen nahezu undenkbar, obwohl es sowohl in Flash- als auch JavaScript möglich ist, barrierefreie Seiten zu bauen, was jedoch ein hohes Maß an Disziplin und Fachwissen des Entwicklers erfordert.

4 Quellen

- [1] William Grosso, *today.java.net*
- [2] Laszlo Systems, *openlaszlo.org*
- [3] Adobe Systems, *adobe.com*
- [4] Ralf Roletschek, *wikipedia.de*
- [5] Wikipedia, *wikipedia.de*