Entwurf und Implementierung einer zeitleistenbasierten Parametersteuerung in eine datenstromorientierte Echtzeitprogrammierumgebung

Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Informatiker

an der Fachhochschule für Technik und Wirtschaft Berlin
Fachbereich Wirtschaftswissenschaften II
Studiengang Angewandte Informatik

Erstbetreuer: Prof. Dr. Thomas Jung
Zweitbetreuer: Dipl. Inf. Sebastian Oschatz

Eingereicht von: Ingolf Heinsch am 10. April 2007

Danksagung

Mein Dank gilt in erster Linie Prof. Dr. Thomas Jung und Dipl. Inf. Sebastian Oschatz. Ohne ihre umfassende Unterstützung und Betreuung wäre diese Diplomarbeit nicht möglich gewesen.

Besonders bedanken möchte ich mich auch bei Jörg Dießl und Sebastian Gregor die mir jederzeit tatkräftig zur Seite standen und immer ein offenes Ohr für Fragen hatten.

Ohne die Hilfe meiner Eltern und Stefan Bleihauers bei der Korrektur, Organisation und Strukturierung wäre diese Diplomarbeit mit Sicherheit nicht rechtzeitig fertig geworden.

Abstract

Diese Arbeit untersucht die Möglichkeit der Kombination einer datenstromorientierten Echtzeitprogrammierumgebung und dem Konzept einer zeitleistenbasierten Parametersteuerung. Dazu erfolgt, neben einer theoretischen Abhandlung des Themas, eine prototypische Umsetzung. Als Framework für diese Untersuchung dient die von der Firma Meso entwickelte grafische Programmiersprache VVVV. Dieses System wird um einen zeitleistenbasierten Editor erweitert. Die Arbeit endet mit exemplarischen Anwendungen.

This work analizes the possibility of the combination of a dataflow oriented real time programming environment and the concept of a timeline-based parameter control. In addition to a theoretical reflection on the issue, a prototypical implementation is realized. The framework for this research project is VVVV, a graphical programming language developed by the Meso Company. This system will be enhanced by a timeline-based editor. The work concludes with exemplary applications.

Inhalt

1. Einführung

1.1. Motivation

Aktuelle wie zukünftige Software sieht sich durch den Einzug des Computers in nahezu allen Bereichen des täglichen Lebens beträchtlichen Anforderungen gegenüber gestellt. Immer lauter wird der Ruf nach Lösungen, die auch vom Anwender an bestimmte Aufgaben angepasst werden können und kein Informatikstudium erfordern. Bereits Ende der fünfziger Jahre entstand das Konzept der visuellen Programmierung. Das Programmieren mit grafischen Hilfsmitteln soll dabei die Konstruktion von Software ermöglichen, ohne eine Zeile Code schreiben zu müssen. Das Programm wird visuell modifizierbar.

Abb. 1: Moog Modular, 1963

Abb. 2: Eniac, 1946

Die visuelle Programmierung hat ihren Ursprung in real existierenden Geräten. So waren es z.B. frühe Synthesizer wie der Moog Modular, Telefonvermittlungsstellen oder erste Universalrechner wie der Eniac, die durch ihren modularen Aufbau und ihre Verkabelungstechnik Modell für die Entwicklung einer neuen Programmiermetapher standen.

Seit ihrer Entstehung pflegt die visuelle Programmierung eher ein Nischendasein und hat es nie vollständig in den Massenmarkt geschafft. Jedoch wagt der Online-Riese Yahoo seit kurzem einen Schritt in diese Richtung. Die im Februar 2007 vorgestellte Software Pipes soll es ermöglichen Webinhalte zu bündeln. Pipes beruht dabei auf der Metapher eines interaktiven Datenstromdiagramms. Doch

auch zuvor gab es besonders im künstlerischen Bereich, aber auch in der Forschung, visuelle Programmiersysteme die sich durchgesetzt haben. Zwei der ersten wirklich erfolgreichen Systeme die auch heute noch genutzt werden, sind Labview und Max[1]. Im Jahr 1986 veröffentlicht, ist Labview ein Programm, welches in der Mess- und Automatisierungstechnik eingesetzt wird und eine eigene grafische Programmiersprache mit dem Namen G besitzt. Ebenfalls im Jahr 1986 erschien das von Miller Puckette, am Pariser Ircam Institut entwickelte Max (damals noch Patcher), eine Programmierumgebung, die es speziell Musikern ermöglicht ihre eigene Software grafisch zu konstruieren.
Ein verwandtes System, dessen Fokus auf der Generierung von Grafiken und Animationen, sowie der Mensch-Maschine Kommunikation liegt, ist das von der Firma Meso entwickelte VVVV. Mit dieser Entwicklungsumgebung, der ebenfalls ein visualisierter Datenstrom zu Grunde liegt, soll es auch Anwendern ohne Informatikstudium möglich sein eigene Softwareideen zu realisieren. VVVV verfolgt dabei einen Echtzeit-Ansatz, d.h., das Programm wird bereits während der grafischen Konstruktion kompiliert[2] und ausgeführt. Jede Änderung wirkt sich umgehend auf das Ergebnis aus. Das macht es gerade für den gestalterischen Einsatz interessant, da kontinuierlich eine visuelle Rückmeldung erfolgt und das Programm damit eher retour, anhand des Ergebnisses, modifiziert werden kann.

Werden Animationen oder grafische Benutzerschnittstellen erstellt, ist es oft wünschenswert, einen zeitlichen Ablauf der Änderungen festzulegen und diesen dann beliebig modifizieren zu können. Herkömmliche Animations- oder Videoschnittsysteme, die weder auf grafischer Programmierung noch auf einem Echtzeitansatz beruhen, bieten zu diesem Zweck Editoren auf Basis einer Zeitleiste an.

[1] Max wird nur noch im Verbund mit dem Zusatzpacket Msp vertrieben und ist daher besser unter dem Namen Max/Msp bekannt.
[2] Übersetzung in Maschinensprache.

1.2. Zielsetzung

Ziel dieser Arbeit ist es zu untersuchen, inwieweit sich das Konzept der datenstromorientierten Echtzeitprogrammierung mit dem Ansatz einer, hauptsächlich auf einem linearen Konzept aufbauenden, zeitleistenbasierten Parametersteuerung vereinbaren lässt. Zu diesem Zweck soll ein grafischer Editor entworfen werden, der eine zeitleistenbasierte Parametersteuerung ermöglicht. Der Editor soll sich dabei nahtlos in die grafische Echtzeitprogrammierumgebung VVV einfügen, welche das Basisframework dieser Arbeit bildet. Zusätzlich wird untersucht, wie sich das lineare Konzept einer Zeitleiste aufbrechen lässt um die Flexibilität weiter steigern zu können.

Da VVV keine Anwendung sondern eine Programmiersprache ist, kann und soll nicht vorherbestimmt werden, für welchen Zweck die zeitleistengesteuerte Parameterbearbeitung später verwendet wird. Es gilt also ein offenes und atomares System zu entwerfen. Grundlegend soll sich der Editor dabei an den bestehenden Konzepten von VVV orientieren und diese fortsetzen. Um die theoretische Untersuchung zu untermauern soll ein Prototyp entwickelt werden, mit dem es möglich ist innerhalb von VVV Parameter mittels einer grafischen Zeitleiste zu bearbeiten.

2. Theoretische Grundlagen

In diesem Kapitel werden die Grundlagen behandelt, die für das Verständnis der weiteren Arbeit unbedingt erforderlich sind. Zuerst wird veranschaulicht, was unter visueller Programmierung zu verstehen ist und wodurch sich speziell die datenstromorientierte Programmierung auszeichnet. Daraufhin wird die Entstehungsgeschichte, sowie die grundlegende Funktionsweise der Programmiersprache VVVV betrachtet.

2.1. Visuelle Datenverarbeitung

Die folgende Abbildung verdeutlicht die Einordnung visueller Programmiersprachen in den Kontext der visuellen Datenverarbeitung.

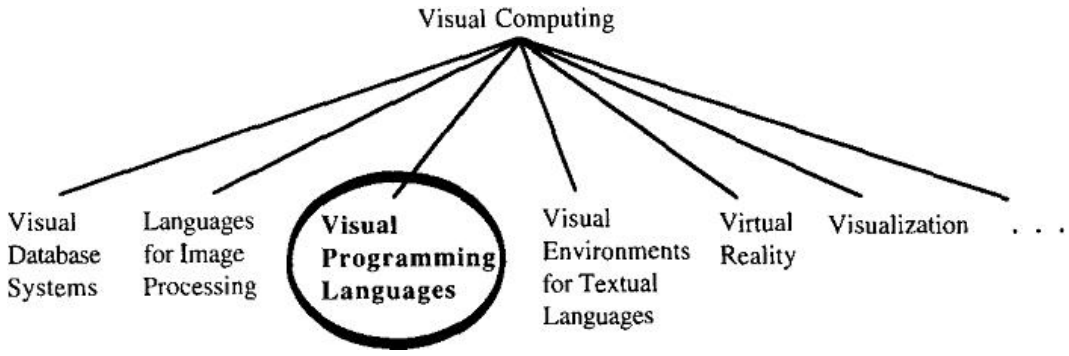


Abb. 3: Einteilung visueller Programmierung nach Burnett [vgl. Burnett94, S.288]

2.1.1. Begriffsanalyse

Trotz langjähriger Existenz der Konzepte „visueller Programmiersprachen“ und „visueller Programmierung“ gibt es bis heute keine eindeutige Definition. Auch ist eine Terminologie im allgemeinen Informatikwortschatz nur selten zu finden.

Daher soll im Folgenden definiert werden, was in dieser Arbeit unter „visueller Programmiersprache“ und „visueller Programmierung“ zu verstehen ist.

Systemen mit visuellen Ansätzen lassen sich in zwei Gruppen einteilen. Visuelle Programmiersysteme mit verbaler- oder grafischer Programmiersprache.

2.1.2. Visuelle Programmiersysteme mit verbaler Programmiersprache

Die markanteste Kennzeichnung verbaler (textueller) Programmiersprachen ist ihre Eindimensionalität. In Bezug auf die Richtigkeit der Semantik[3] wird lediglich die Abfolge der Zeichen überprüft. Für den *Compiler*[4] ist die grafische Anordnung eines Programmcodes meist unsichtbar. Ein Programm wird nur anhand korrekter Syntax[5] kompiliert [vgl. Brunett99, S.1].

```
public void MouseDownRight(int x, int y){if ((x > 0 && x < 0 + cWidth)
&& (y > cYPos && y < cYPos + cHeight)){double sqrt = 0;for (int i=0;
i<graph.Points.Count-1; i++){for (int
j=0;j<graph.PCount;j++){sqrt=Math.Sqrt(Math.Pow(graph.Points[i][j].X-
x,2)+Math.Pow(graph.Points[i][j].Y - y,2));if(sqrt < 5)FSelected.X =
i;FSelected.Y = j;return;}}}}}
```

Listing 1: Syntaktisch korrekter Quellcode

Der in Listing 1 gezeigte Programmcode ist zwar syntaktisch korrekt, aber so amorph, dass er für den Menschen kaum lesbar ist. Der Visualisierungsgrad ist kaum messbar, da dieser Text keine visuellen Hinweise auf den logischen Aufbau enthält.

```
public void MouseDownRight(int x, int y)

{

    if ((x > 0 && x < 0 + cWidth) && (y > cYPos && y < cYPos + cHeight))

   {

       double sqrt = 0;

       for (int i=0; i<graph.Points.Count-1; i++)

      {

          for (int j=0; j<graph.PCount; j++)

          {

            sqrt = Math.Sqrt(Math.Pow(graph.Points[i][j].X - x,2) +
```

[3] Die Bedeutung von Zeichen.

[4] Programm, welches Quelltext in Maschinensprache übersetzt.

[5] Beziehung der Zeichen untereinander.

```
            Math.Pow(graph.Points[i][j].Y - y,2));
        if (sqrt < 5)
        {
             FSelected.X = i;
             FSelected.Y = j;
             return;
        }
        }
      }
   }
}
```

Listing 2: Syntaktisch korrekter Quellcode mit visueller Anordnung

Die Lesbarkeit des Programmcodes erfolgt bei textuellen Programmiersprachen durch das Hinzufügen von visuellen Elementen. Listing 2 zeigt den aus Listing 1 bekannten Quellcode in visuell lesbarer Form.

2.1.3. Visuelle Programmiersysteme mit grafischer Programmiersprache

Den Kern grafischer Programmiersprachen bildet die, bei textuellen Programmiersprachen fehlende, visuelle Semantik. Bei grafischen Programmiersprachen haben die visuellen Elemente eine programmiertechnische Bedeutung. Es wird eine weitere Dimension, z.B. die räumliche Anordnung von Elementen zur Darstellung semantischer Beziehungen hinzugefügt. Als Beispiel für eine solche Beziehung könnte man sich Objekte vorstellen, die durch Linien miteinander verbunden sind um deren Abhängigkeit voneinander zu visualisieren.

Eine visuelle Programmiersprache dient der Beschreibung von Software. Kann mit dieser jedes Berechnungsverfahren formuliert werden, welches auf einer Turing-Maschine[6] ausführbar ist, liegt dem eine Universalprogrammiersprache zu Grunde. Da alle berechenbaren Probleme mittels einer Turing-Maschine lösbar sind, bezeichnet man visuelle Universalprogrammiersprachen als berechnungsuniversell [vgl. Schiffer98, S.64f].

[6] Mathematisches Modell mit nur drei Operationen.

Als visuelle Programmierung wird die Erstellung von Software mit Hilfe von grafischen Programmiersprachen bezeichnet. Programmierumgebungen mit grafischen Benutzungsschnittstellen für verbale Programmiersprachen, z.B. SharpDevelop[7], die mit WISIWYG[8] Editoren arbeiten, werden nicht als visuelle Programmierung verstanden [Schürr01, S.14]. Die Menge an integrierten Werkzeugen zur Programmierung bildet ein visuelles Programmiersystem. In den meisten Fällen bildet die Benutzerschnittstelle eines solchen Systems einen Teil der zu Grunde liegenden Sprache ab [vgl. Shu88, S.12].

2.1.4. Klassifikation visueller Programmiersysteme

Da, wie in Kapitel 2.1.3. beschrieben, eine Trennung zwischen visuellen Programmiersprachen und deren Werkzeugen nicht immer möglich ist, werden sie im Folgenden anhand vorhandener Sprachkonzepte klassifiziert. Es existieren verschiedene Systeme nach denen sich visuelle Programmiersysteme klassifizieren lassen, etwa das von Nan C. Shu [vgl. Schiffer98, S.111f].

Die meisten Systeme beruhen auf der Unterscheidung von Anwendungsfällen oder der Art der Visualisierung. Die im Folgenden verwendete Klassifizierung erfolgt nach Brunett und Baker [vgl. Brunett94, S.287-300]. Das bietet die Möglichkeit, visuelle Programmiersysteme anhand ihrer zu Grunde liegenden Paradigmen[9] zu klassifizieren.

Grundlegend können diese in drei Gruppen klassifiziert werden: [vgl. Schiffer98, S. 131]

- Basierend auf verbalen Programmiersprachen:
  steuerflussorientierte-, funktionsorientierte- ,objektorientierte, constraintorientierte- und datenstromorientierte- visuelle Programmiersysteme.
- Basierend auf grafischen Programmiersprachen:
  regelorientierte-, beispielorientierte- und formularorientierte visuelle Programmiersysteme
- Visuelle Programmiersysteme für Spezialfälle:
  multiparadigmenorientierte visuelle Programmiersysteme

Da dieser Arbeit ein datenstromorientiertes Programmiersystem als Basisframework dient, wird dieses System nachfolgend genauer erläutert.

---

[7] Freie Entwicklungsumgebung für die Programmiersprache C#.

[8] Abkürzung für das Prinzip "What you see is what you get". Das zu bearbeitende Dokument wird genau so angezeigt, wie es auf einem späteren Ausgabegerät aussieht.

[9] Die Grundprinzipien einer Programmiersprache.

2.1.5. Datenstromorientierte Programmiersysteme

Datenstromorientierte Programmiersysteme (auch datenflussorientierte Programmiersysteme) basieren auf gerichteten Graphen, innerhalb dieser werden die Funktionen und Operationen als Knoten und Datenkanäle als Kanten dargestellt. Über diese Datenkanäle kommunizieren die einzelnen Knoten in einem datenstromorientierten Programmiersystem. Diese Systeme verfügen weiterhin über Datenquellen und Datensenken, über die Daten ein- und ausgegeben werden können.

Dieses Paradigma bildet die theoretische Basis der visuellen Programmiersprache VVVV. Im Folgenden wird genauer auf dieses Paradigma eingegangen. Datenstromorientierte Programmiersysteme basieren auf Datenflussdiagrammen.

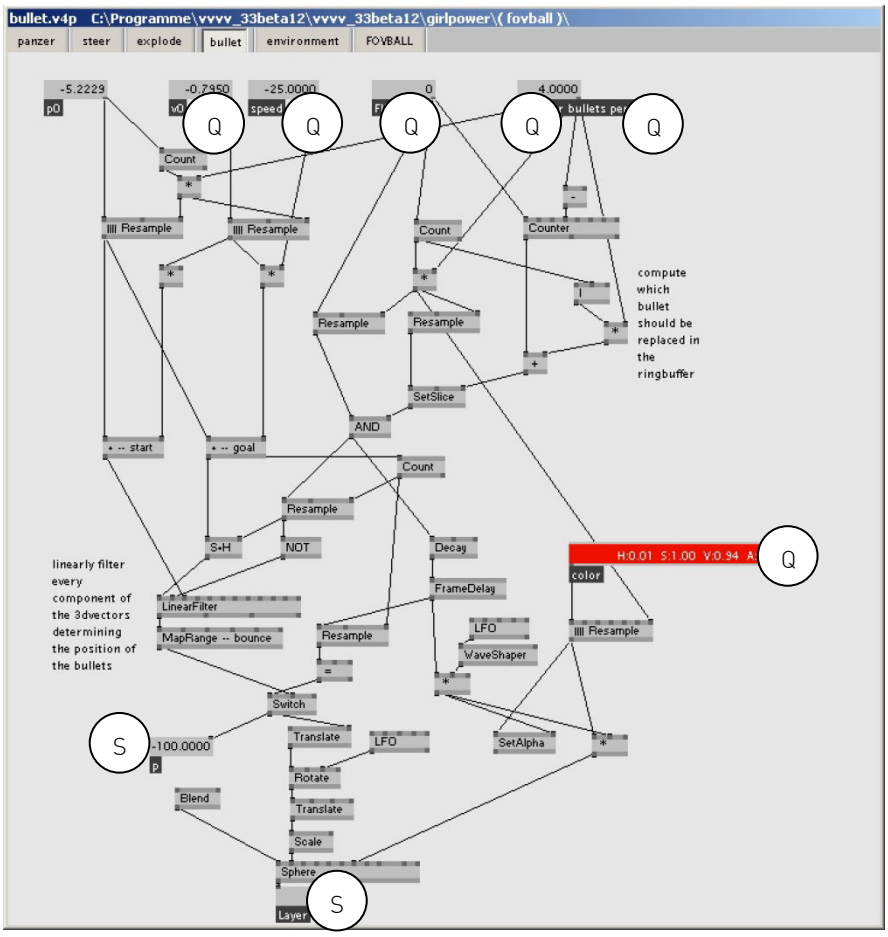


Abb. 4: Datenstrom in VVVV

In Abbildung 4 sind Datenquellen mit einem *Q*, Datensenken mit einem *S* markiert. Die restlichen Knoten beinhalten Operationen auf diesen Daten. Datenquellen können hierbei auf konstanten Werten oder Wertgeneratoren, wie

z.B. der mathematischen *random*-Funktion[10] basieren. Knoten, die Operationen beinhalten, können selbst wiederum aus Datenflussdiagrammen bestehen.

Visuelle Datenstrom-Diagramme unterliegen verschiedenen Regeln, die für die Verknüpfung von Datenquellen, Datensenken und der Knoten gelten.

Diese sind:

- Pro Eingang darf nur ein Datenkanal angeschlossen sein.
- Pro Ausgang können mehrere Datenkanäle angeschlossen werden.
- Datenquellen besitzen ausschließlich Ausgänge.
- Datensenken besitzen ausschließlich Eingänge.
- Ein Knoten besitzt mindestens einen Eingang und keinen oder mehrere Ausgänge.
- Jeder Eingang eines Knotens muss mit einem Datenkanal verbunden sein oder eine Datenquelle besitzen. Sonst ist die Operation nicht ausführbar[11].
- Ein Knoten ohne Ausgänge oder Verbindungen zu anderen Knoten oder Datensenken ist wirkungslos.

Einige Systeme, wie z.B. Max, befolgen aber nicht strikt alle diese Regeln und brechen damit das Kernkonzept datenstromorientierter Programmierung auf. Im Vergleich zur funktionalen Programmierung können die Eingänge eines Knotens als Funktionsparameter, die Operation als die eigentliche Funktion und die Ausgänge als Rückgabewert dieser Funktion betrachtet werden.

Ausführung
Operationen werden in datenstromorientierten Programmiersystemen ausgeführt, sobald Eingabedaten zur Verfügung stehen. Um Ausgabedaten berechnen zu können, werden zunächst alle Eingangsdaten gesammelt. Die Operation wird auf diesen Daten ausgeführt und ein Datenwert pro Ausgang zur Verfügung gestellt. Ein Ausgang darf hierbei immer nur einen Datenwert erhalten, damit die folgende Verarbeitung aller Daten gewährleistet ist. Datensenken und Knoten akzeptieren als Eingang immer nur einen Datenwert [vgl. Schiffer98, S.143f].

2.1.5.1. Merkmale

Datenstromorientierte Programmierung besitzt einige entscheidende Besonderheiten. Die Wichtigsten werden kurz vorgestellt.

[10] Die Funktion *random* generiert Zufallswerte.
[11] In manchen Systemen wird diese Regel, durch Generierung von Standardwerten ersetzt.

Parallelität und implizite Ausführungsreihenfolge
Bei der datenstromorientierten Programmierung gibt der Programmierer vor, wann Eingangsdaten in Operationen eingegeben werden. Anstatt die Reihenfolge der Verarbeitung von Knoten zu bestimmen, legt der Programmierer durch das Erstellen von Kanten Abhängigkeiten zwischen Knoten fest. Hierfür werden Ausgänge von Knoten oder Datenquellen mit Eingängen von anderen Knoten oder Datensenken verbunden. Da datenverarbeitende Knoten immer selbst die Daten anfordern, ist eine implizite Verarbeitungsreihenfolge gegeben. Wenn eine Reihenfolge von Operationen gewünscht ist, die nicht voneinander abhängig sind, kann dieses Modell zu Problemen führen. Hierfür werden explizite Sprachkonstrukte bereitgestellt, die eine sequentielle Verarbeitung von Knoten ermöglichen. Da alle Operationen, die nicht voneinander abhängig sind, parallel ausgeführt werden, muss sich der Programmierer um die parallele Ausführung nicht kümmern [vgl. Schiffer98, S.145].

Nebeneffekte
Als Nebeneffekte werden zeitabhängige und nicht-lokale Wirkungen bezeichnet. Diese existieren in einem Datenstrommodell nicht. Das bedeutet allerdings auch Einschränkungen in der Programmiervielfalt. Zustandsvariablen und die Nutzung globaler Systemvariablen sind beispielsweise für die Interprozesskommunikation[12] auf diese Nebeneffekte angewiesen [vgl. Schiffer98, S.145f].

Variablen
Da in der datenstromorientierten Programmierung die Daten immer auf den Datenkanälen gespeichert sind, werden keine Variablen benötigt. Ein Datenkanal muss nicht benannt werden und ist visuell sofort erfassbar. Das ist wahrnehmungsphysiologisch ein großer Vorteil gegenüber anderen Systemen. Sobald ein Knoten eine Operation ausgeführt hat, liegt das Ergebnis am Ausgang des Knotens und damit an dem verbunden Datenkanal vor. Es kann daher direkt von dem abhängigen Knoten oder der angeschlossenen Datensenke verwendet werden. Zudem stellt es die implizite Ausführungsreihenfolge sowie die beschriebene Nebeneffektfreiheit sicher [vgl. Ackermann82, S. 15f].

Verzweigungen und Schleifen
Um Ablaufstrukturen wie *IF*, *THEN*, Switches, oder Schleifen zu realisieren, sind spezielle Konzepte notwendig. Daher müssen Konstrukte für die iterative Abarbeitung erweiterter Regeln und die Vorbelegung von Datenkanälen möglich sein, um diese Anweisungen zu realisieren.

[12] Methoden zum Informationsaustausch zwischen Prozessen.

2.2. VVVV

Im Folgenden wird erläutert, wie VVVV entstanden ist und für welche Anwendungsfälle es entwickelt wurde. Im weiteren Verlauf des Kapitels wird auf die Benutzerschnittstelle und die grundlegende Funktionsweise näher eingegangen.

„vvvv is a toolkit for real time video synthesis. It is designed to facilitate the handling of large media environments with physical interfaces, real-time motion graphics, audio and video that can interact with many users simultaneously." [vgl. VVVV07]

2.2.1. Entstehung und Anwendung

Das Design-Büro Meso - digital interiors ist spezialisiert auf den Entwurf, die Gestaltung und die Umsetzung von Arbeiten mit interaktiven, digitalen Medien. Im Querschnitt finden sich hier sowohl experimentelle, als auch angewandte Arbeiten mit dem klaren Fokus, den Benutzer auf spielerische Weise mit komplexer Technik umgehen zu lassen. Im Jahr 1998 wurde bei Meso mit der Entwicklung einer Software begonnen, die einen hohen Grad an programmiertechnischer Komplexität gleichberechtigt mit einer gestalterischen Arbeitsweise verbinden sollte. Ein weiteres Primärziel der Entwicklung war es, hardwarebeschleunigte Grafik in einem designorientierten Kontext zugänglich zu machen. Zusätzlich sollte die programmiertechnische Komplexität soweit verborgen werden, dass es auch Anwendern ohne langjährige Programmiererfahrung ermöglicht wird, eigene Ideen realisieren zu können. Zunächst als Komponentensammlung genutzt, entstand 2001 im Rahmen einer Diplomarbeit das grafische Benutzer-Interface des Datenstroms [vgl. Dieß01].

Das, in der Programmiersprache Delphi[13] entwickelte VVVV ist nach Kapitel 2.1.5. ein datenstromorientiertes visuelles Programmiersystem.

Hauptanwendungsgebiet von VVVV sind interaktive Medieninstallationen - ein weitläufiger Begriff, der anhand von Beispielen eingegrenzt wird.

Häufig findet VVVV Einsatz in der experimentellen Medienkunst. Hierbei werden z.B. physische Aktionen des Benutzers mittels Sensoren erfasst und die so gewonnenen Daten grafisch umgesetzt, also ein interaktives System entwickelt. Abbildung 5 zeigt ein Foto der Performance Imago[14], die mit VVVV entwickelt wurde. Bei dieser Performance wurde ein 3D Modell des Akteurs anhand seiner Stimme modifiziert.

---

[13] Eine von der Firma Borland entwickelte objektorientierte Programmiersprache.
[14] Imago entstand im Team um Dietmar Bruckmayr.

Abb. 5: Interaktive Performance; Mittels akustischer Signale wird ein 3D Modell des Künstlers modifiziert

Meso setzt VVVV auch für kommerzielle Projekte ein. So werden z.B. Medien-Installationen für Messen, Museen, sowie medial unterstützte Architekturen realisiert. Abbildung 6 zeigt eine Installation, welche zur Hauptvollversammlung der BASF AG im Jahr 2006 im Mannheimer Rosengarten zu sehen war. Hierfür wurde eine 360° Vollkugelprojektion entwickelt.

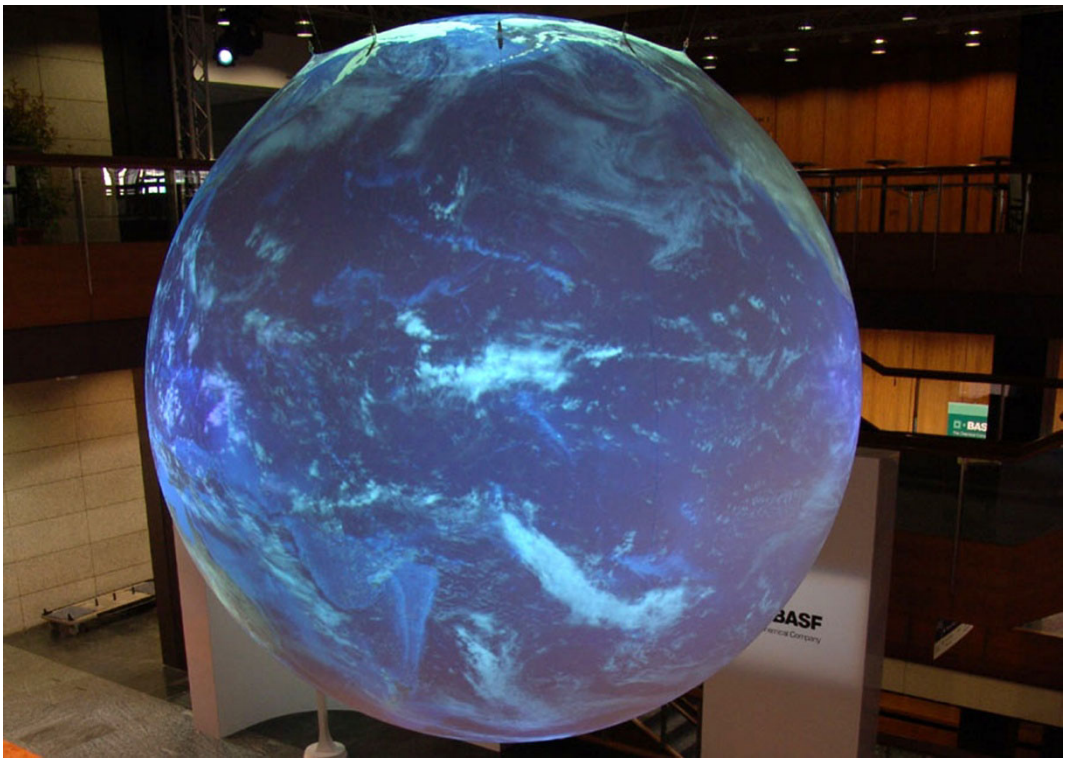


Abb. 6: Frei hängende Kugel, auf die mit 6 Projektoren eine animierte Erdoberfläche projiziert wird

Ein weiterer Anwendungsfall ist das Prototyping[15]. Als Echtzeitanwendung besitzt VVV auch auf dem diesem Gebiet klare Vorteile. Jede Änderung am Programm erzielt umgehend ein Ergebnis. Durch das Zeichnen eines Datenstroms lassen sich sehr schnell komplexe Algorithmen entwerfen, die entweder direkt in VVV verfeinert- oder in textuelle Programmiersprachen umgesetzt werden können.

2.2.2. Benutzerinterface und Basiskonzepte

Wird VVV ausgeführt, startet es wahlweise mit einem Beispielprogramm (in VVV und im nachfolgenden „Patch" genannt) oder mit einem neuen, leeren Patch. Wird ein neuer Patch erstellt, so präsentiert sich VVV dem Benutzer mit einem schlichten, grauen Fenster. Lediglich der Dateiname des Patches wird dem Benutzer angezeigt (Abb. 7). Es existiert kein statisches Menü, wie es von gängiger Software bekannt ist. Das gewährleistet besonders bei komplexen Patches eine größtmögliche Fokussierung auf den eigentlichen Zweck des Programms. Alle Interface-Elemente, die der Benutzer für sein Programm benötig, wird er selbst erstellen.

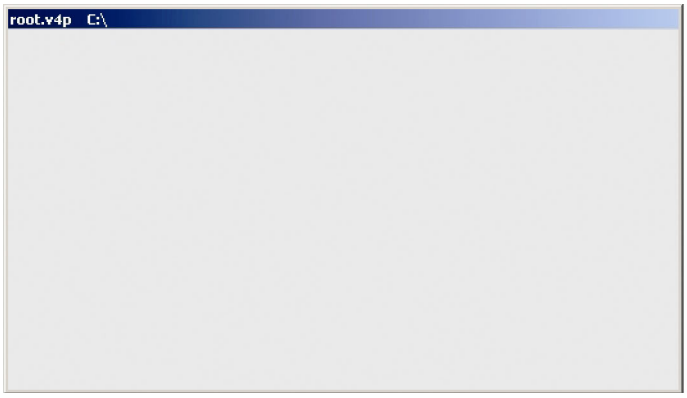


Abb. 7: VVV zeigt nach dem Starten ein leeres Patch-Fenster

Im Gegensatz zu konventionellen Programmiersprachen existiert nur ein Modus: der Laufzeitmodus. Es ist kein Kompilieren nötig. Jede Änderung wirkt sich sofort auf das Programm aus.

VVV ist eine Echtzeitprogrammierumgebung.

Weiterhin besteht die Möglichkeit beliebig viele Patches zu erstellen. Diese laufen alle gleichzeitig ab und können untereinander Daten austauschen. Um mit VVV zu interagieren, existieren verschiedene Tastenkombinationen und drei verschiedene Menüs. Das Hauptmenü, das z.B. zum Laden und Speichern von

[15] Das Prototyping ist eine Methode der Softwareentwicklung, die ein frühzeitiges Feedback bezüglich der Eignung eines Lösungsansatzes ermöglicht.

Patches dient, sowie das Knoten- und Inspektormenü, die beide im folgenden Abschnitt erläutert werden.

2.2.3. Knoten, Kanten, Pins

Wie in Kapitel 2.1.5. beschrieben, setzt ein datenstromorientiertes Programmiersystem Datenquellen, Datensenken und Operationen voraus. Diese werden in VVV durch Knoten abgebildet. Knoten repräsentieren in VVV sowohl Datenquellen, als auch Datensenken, da sie Eingabe- und Ausgabeparameter besitzen können. Grundlegend wird in zwei Typen von Knoten unterschieden: funktionale Knoten und Ein– und Ausgabeknoten. Die Gruppe der funktionalen Knoten führt Operationen aus.

Verglichen mit der textuellen Programmierung übernimmt ein funktionaler Knoten die Rolle einer Funktion. Er erhält einen- oder mehrere Eingabe-Parameter, führt anhand dieser eine oder mehrere Operationen aus und gibt einen- oder mehrere Ausgabeparameter zurück. Aufgrund dessen ist es, im Gegensatz zur textuellen Programmierung vergleichsweise einfach, Funktionen mit mehreren Ausgabeparametern zu versehen und diese an verschiedenen Stellen im Programm weiter zu verwenden.

Die zweite Gruppe von Knoten bilden die Ein- und Ausgabeknoten (in VVV und im Nachfolgenden „IO-Box“ (Input / Output Box) genannt). Dieser Knotentyp dient zum Erstellen und Ausgeben von Daten, beschreibt also nach dem datenstromorientierten Programmiermodell aus Kapitel 2.1.5. Datenquellen und Datensenken. IO-Boxen existieren für jeden in VVV verfügbaren Datentyp. Auf die vorhandenen Datentypen wird in Kapitel 2.2.4. näher eingegangen. IO-Boxen dienen maßgeblich zur Gestaltung von Benutzerschnittstellen. Deshalb werden sie in Kapitel 3.1.2. noch einmal ausführlich behandelt.

Die Ein- und Ausgabeparameter eines Knotens werden durch Pins repräsentiert. Von diesen kann jeder Knoten, je nach Funktion, beliebig viele besitzen. Da VVV einen Datenstrom abbildet, in welchem die Information von der Datenquelle (oben) zur Datensenke (unten) fließt, befinden sich die Pins der Eingabeparameter (in VVV Input-Pins) immer an der Oberseite eines Knotens und die Pins der Ausgabeparameter (in VVV Output-Pins) immer an der Unterseite eines Knotens (Abb. 8).

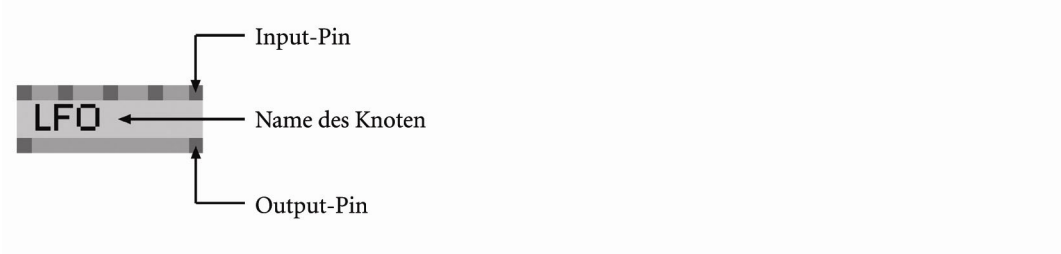


Abb. 8: Knoten in VVV

Die Abbildung der aus dem datenstromorientierten Programmiermodell bekannten Kanten, erfolgt durch das Verbinden von Output mit Input-Pins. Output-Pins eines Knotens lassen sich typsicher[16] mit Input-Pins anderer Knoten verbinden. Die Typsicherheit wird zur Laufzeit überprüft. Es werden nur typsichere Verbindungen zugelassen und so undefinierte Zustände vermieden. Hierbei kann jeder Output-Pin mit beliebig vielen Input-Pins verbunden werden (*1:n* Beziehung), aber jeder Input-Pin darf nur eine Verbindung aufweisen (*1:1* Beziehung). Auch das wird von VVVV zur Laufzeit überprüft. Abbildung 9 zeigt einen Patch, der eine Addition zweier Zahlen ausführt.

Durch die Überprüfung der Typsicherheit werden Programmierfehler ausgeschlossen.

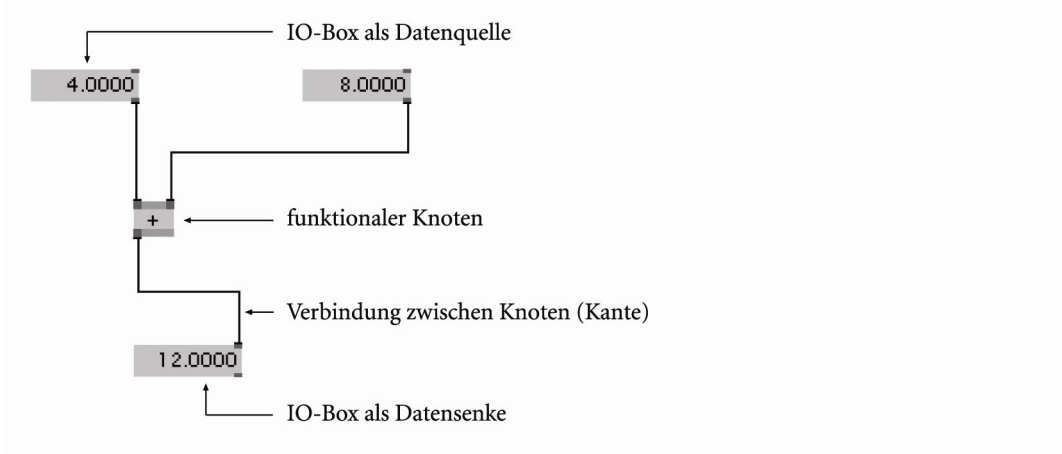


Abb. 9: Einfaches Additionsprogramm in VVVV

VVVV bietet dem Benutzer eine Vielzahl von Knoten an. Diese verfügen von simplen, mathematischen Funktionen, über den Zugriff auf die Hardware des Computers (z.B. die Ansteuerung der seriellen Schnittstelle), bis hin zu Netzwerkprotokollen über alles, was benötigt wird um selbst komplexe Softwarelösungen zu realisieren. Um einen Knoten zu erstellen bietet VVVV je nach Kenntnisstand verschiedene Wege an. Zum einen kann ein Knoten direkt durch die Eingabe seines Namens erstellt werden. Diese Funktionalität nutzen aber eher fortgeschrittene Nutzer. Den leichteren Einstieg ermöglicht das Knotenmenü. Hier werden alle vorhandenen Knoten aufgelistet (Abb. 10).

[16] Typsicherheit bezeichnet den Zustand (einer Programmausführung) bei dem die Datentypen, gemäß ihren Definitionen in der benutzten Programmiersprache, verwendet werden.

Abb. 10: Alphabetisch sortiertes Knotenmenü von VVV

Nach Erstellen eines Knotens, ist dieser u.U. an die Programmbedürfnisse anzupassen. Alle Basisinformationen, z.B. über den Typ der In- und Output-Pins sind durch Tooltips direkt am Knoten ablesbar und können auch dort verändert werden. Um Komplexität zu verbergen verfügt VVV über ein weiteres Menü, welches sämtliche Eigenschaften eines Knotens anzeigt - das Inspektormenü (Abb. 11).

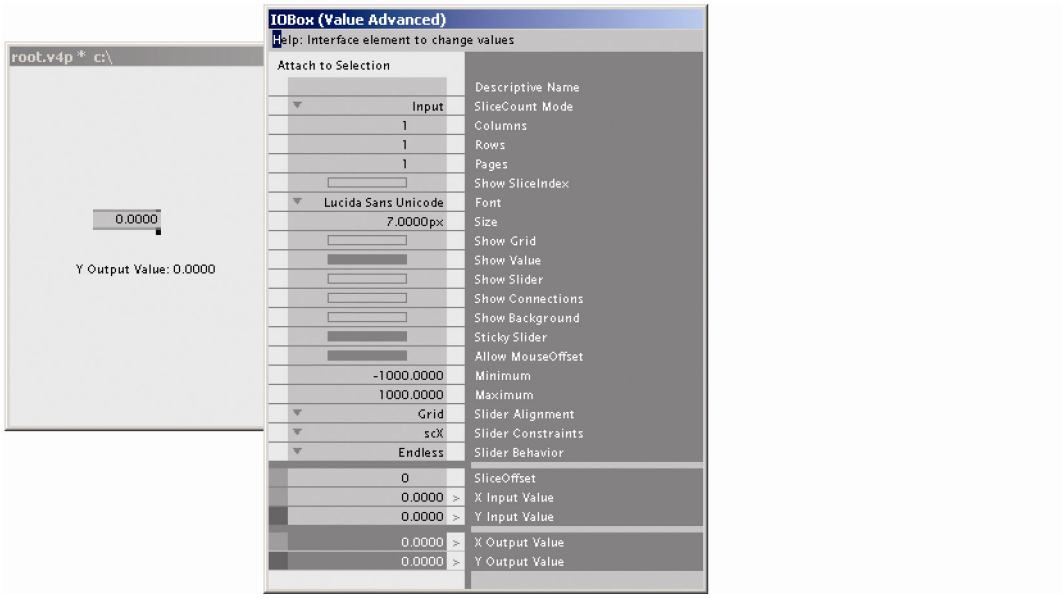


Abb. 11: Das Inspektormenü zeigt alle Eigenschaften des Knotens an

2.2.4. Datentypen

Wie textuelle Programmiersprachen auch, kennt VVVV verschiedene Datentypen. Grundlegend werden fünf Datentypen unterschieden [vgl. VVVV07].

- *Value*
  Dezimalzahlen des Intervalls [*min.double*, *max.double*]. Daneben existieren verschiedene Subtypen wie z.B. Booleans oder zyklische Werte (für Winkelangaben).
- *String*
  Zeichenketten im Ascii oder Unicode UTF-8[17] Format.
- *Color*
  Dieser Datentyp erlaubt es dem Benutzer mit Farben zu arbeiten, unabhängig von einem spezifischen Farbmodell. Es stehen verschiedene Knoten zur Auswahl, die Farbmodelle abbilden. Während der Erstellung eines Programms muss sich der Benutzer aber nicht um die Funktionsweise eines Farbmodells kümmern.
- *Enum*
  Aufzählungen.
- *Node*
  Ein Abstrakter Datentyp, der zur Beschleunigung der Graphberechnung von VVVV dient. Nodetypen sind z.B.: Transformationen, Renderstates, Texturen, Vertexbuffer[18] und DirectShow[19] Audio- und Videodaten.

Jeder Pin eines Knoten repräsentiert immer einen Datentyp. Aus diesem Grund können an einen Pin des Typs *value* auch nur Daten des gleichen Typs angelegt werden.

2.2.5. Grafikausgabe / Rendering

VVVV besitzt eine Vielzahl von Möglichkeiten zur visuellen Interaktion. Neben dem Patch-Fenster existieren verschiedene Typen von Ausgabefenstern (in VVVV und im Nachfolgenden „Renderer" genannt). Der wichtigste Renderer-Typ ist der Direct-X Renderer. Dieser und die ihm zugehörigen Knoten beruhen auf der Direct-X API. Weitere Renderer-Typen sind z.B.: GDI, Flash, HTML oder auch TTY. Darüber hinaus bietet VVVV einen textuellen Editor zur Entwicklung von Pixel-

[17] Weit verbreiteter Unicodestandard. Beinhaltet fast sämtliche Alphabete.
[18] Speicher für Geometriedaten.
[19] Teil der Direct-X API. Dient zur Verarbeitung von Audio- und Videodaten.

oder Vertex-Shadern[20]. Dieser erlaubt es HLSL[21] Shader-Programme zu schreiben.

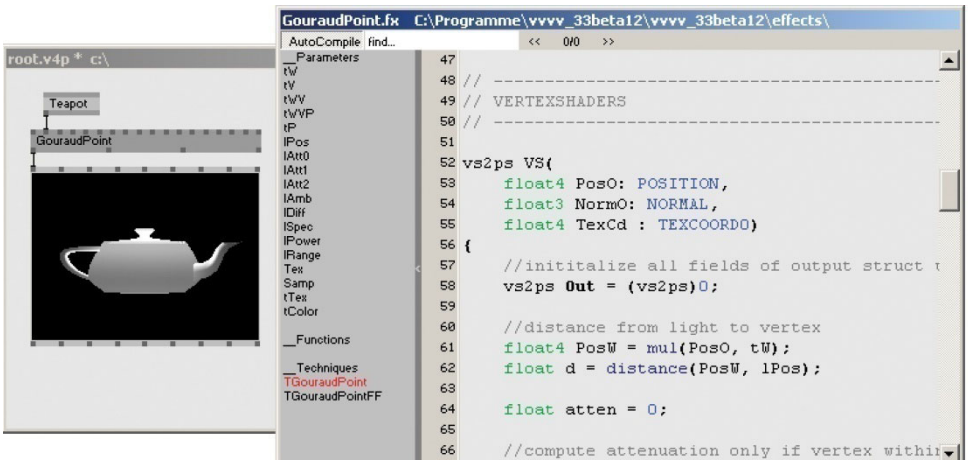


Abb. 12: Pixel- und Vertex-Shader Editor in VVV

2.2.6. Datenverarbeitung

Eines der wesentlichen Programmierkonzepte von VVV ist das Verarbeiten von Listen (in VVV und im nachfolgenden „Spreads" genannt). Spreads sind eindimensionale Listen, welche aus n-Elementen (in VVV und im nachfolgenden „Slices" genannt) bestehen. Ein Großteil der in VVV vorhandenen Knoten akzeptiert Spreads. Das Konzept der Spreads ermöglicht beispielsweise das Instanziieren von grafischen Objekten ohne den Knoten, der das Objekt erzeugt, selbst vervielfältigen zu müssen. Dieses wird anhand eines Beispiels deutlicher.

Es soll z.B. fünfmal ein grafisches Objekt (ein Rechteck) erzeugt werden.

Die Abbildung 13 zeigt die Lösung der Aufgabe ohne die Verwendung von Spreads. Es muss fünfmal der Knoten *Quad*, welcher ein Viereck zeichnet, erzeugt werden. Zudem erhält jeder Knoten eine Translation auf der X-Achse (durch den Knoten *Transform*). Danach werden alle fünf Elemente gruppiert und auf einem Direct-X Renderer ausgegeben.

[20] Pixel– und Vertex-Shader sind Programme, die vom Grafikprozessor einer 3D-Grafikkarte im Verlauf der Rendering-Pipeline ausgeführt werden.

[21] HLSL (High Level Shading Language) ist eine Programmiersprache, die zur Entwicklung von Pixel– und Vertex-Shadern dient.

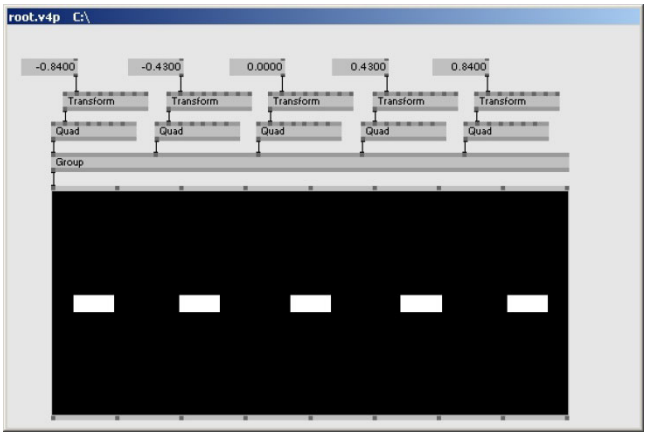


Abb. 13: Instanziierung von Objekten ohne die Verwendung von Spreads

Ohne die Verwendung von Spreads werden für das Zeichnen von fünf Vierecken siebzehn Knoten benötigt. In Abbildung 14 wird dieselbe Aufgabe mit einem Spread gelöst, der eine Liste von Translationen an einen Transformations-Knoten übergibt. Dadurch wird das Objekt instanziiert. Für jeden Slice entsteht ein neues Objekt.

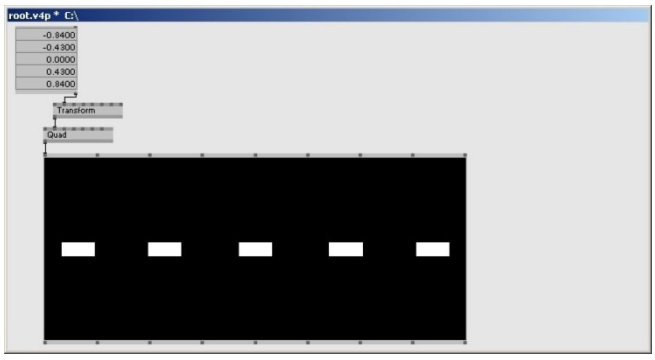


Abb. 14: Instanziierung von Objekten durch Spreads

Mit Hilfe von Spreads kann die Anzahl der benötigten Knoten deutlich reduziert werden. Neben der Möglichkeit die gewünschten Werte innerhalb eines Spreads manuell zu erstellen, stellt VVVV eine Reihe von Spread-Generatoren zur Verfügung. Diese Knoten erzeugen dynamisch Spreads, die unter Verwendung von verschiedenen Parametern beeinflusst werden können. Das einfachste Beispiel hierfür ist der Knoten *LinearSpread*, der eine Reihe von Werten anhand eines Startwertes, der Schrittweite und anderen Parametern erzeugt.

Um ein einzelnes Slice (Element) aus einer Liste auszulesen oder zu verändern existieren die Knoten *GetSlice* und *SetSlice*. Mit *GetSlice* wird unter Angabe eines Index ein bestimmter Slice ausgegeben. *SetSlice* kann wiederum ein Slice innerhalb eines Spreads verändern.

2.2.7. Kommunikation

Ein weiteres Hauptmerkmal von VVVV stellt die Möglichkeit der Netzwerkkommunikation, sowie die Steuerung externer Geräte dar. Zu diesem Zweck stehen eine Reihe spezifischer Knoten zur Verfügung.

Netzwerkkommunikation
Neben den Standardprotokollen TCP/IP[22], welches ein verbindungsorientiertes Ende-zu-Ende Modell repräsentieren [vgl. TCP05] und UDP[23], ein verbindungsloses Kommunikationsmodell zur Nachrichtenübermittlung [vgl. UDP05], ist besonders das OSC-Protokoll hervorzuheben. Das OSC-Protokoll (Open Sound Control) wurde an der University of California, Berkeley entwickelt und dient in erster Linie zur Kommunikation im Multimediabereich. OSC ist unabhängig von einem Transportprotokoll, es kann also sowohl über TCP, UDP oder auch über die serielle Schnittstelle des Computers versendet werden und wurde mit dem Ziel entwickelt, das veraltete MIDI-Protokoll[24] abzulösen [vgl. Kling02, S.1]. Meistens bildet jedoch das UDP-Protokoll die Basis für OSC. Es liegt im OSI-Referenzmodell[25] auf der Anwendungsschicht. Die Übermittlung von Daten erfolgt mittels OSC-Messages. Diese bestehen aus einem eindeutigen Namen (OSC-Address Pattern[26]), einem Type-Tag, welcher den Datentyp des zu übermittelnden Wertes angibt, sowie dem Wert selbst. OSC-Messages können sowohl einzeln, also auch gesammelt (OSC-Bundles) versendet werden [vgl. Wright02].

Hardware
VVVV verfügt über verschiedene Wege, um mit an den Computer angeschlossener Hardware zu kommunizieren. Dies können sowohl fest als Knoten integrierte Treiber für spezifische Hardware (z.B. der Knoten *Touchscreen*, der zum Auslesen der Daten eines Touchscreen Monitors dient), als auch die Ansteuerung der seriellen Schnittstelle, z.B. zum Auslesen von Sensoren sein [vgl. VVVV07]. Neben den hier beschriebenen Wegen der Kommunikation existieren in VVVV noch weitere, auf die aber im Rahmen dieser Arbeit nicht näher eingegangen wird.

---

[22] Transmission Control Protocol, verbindungsorientiertes, zuverlässiges Netzwerkprotokoll.
[23] User Datagram Protokoll, verbindungsloses, unzuverlässiges Netzwerkprotokoll.
[24] Datenprotokoll zur Steuerung von Musikgeräten.
[25] Beschreibt modellhaft die Datenübertragung zwischen Computern.
[26] Ein String der von einem „/" angeführt wird.

2.2.8. Berechnung

Um zu verstehen was sich hinter dem Begriff Echtzeitsystem verbirgt, wird kurz beschrieben, wie VVVV intern berechnet wird. VVVV teilt dazu die CPU-Zeit in sogenannte Frames auf. Zu einem spezifischen Zeitpunkt werden alle Werte aller Input-Pins in einem Patch gesammelt. Anschließend erfolgt eine Berechnung der Werte. Danach werden diese an die entsprechenden Output-Pins übergeben. In jedem Frame werden sämtliche Knoten berechnet, deren Output-Pins für die Berechnung anderer Knoten notwendig sind. Eine Zuweisung der Ausführungsreihenfolge ist nicht möglich. Da jeder dieser Knoten nur einmal pro Frame berechnet wird, existiert keine Möglichkeit der Schleifenbildung. Es werden im späteren Verlauf aber Konzepte präsentiert, die dieses Manko ausgleichen.

2.3. Zeitgesteuerte Parameter

Da der Begriff Parameter eine zentrale Position einnimmt, folgt eine kurze Erläuterung, was im Rahmen dieser Arbeit darunter verstanden wird.

Als Parameter werden im Folgenden alle nicht statischen Eigenschaften von virtuellen- oder realen Objekten aufgefasst, die der Steuerung durch ein Computerprogramm unterliegen. Das können beispielsweise Eigenschaften eines geometrischen Objektes innerhalb einer 3D-Szene oder eines realen Objektes, z.B. die Drehgeschwindigkeit eines Motors sein. Ein Parameter ist gleichzusetzen mit einer Variablen. Am Beispiel des Aufbaus einer typischen 3D-Szene wird dieses kurz erläutert.

Eine 3D-Szene enthält in der Regel folgende Komponenten, mit dazugehörigen Parametern:

- *Ein- oder mehrere geometrische Objekte*
  Definition durch Punkt- oder Flächenlisten, Objektfarbe, Material, Textur
- *Ausrichtung und Skalierung der Objekte innerhalb des Weltkoordinatensystems*
  Translation, Skalierung, Rotation, Objektmittelpunkt für jede Achse
- *Eine- oder mehrere Lichtquellen*
  Typ der Lichtquelle, Farbe des Lichts, Intensität
- *Betrachter (Kamera)*
  Ausrichtung und Blickrichtung innerhalb des Weltkoordinatensystems, Brennweite
- *Ausgabe*
  Art der Projektion, Sichtvolumen, Farbtiefe

Durch die Vergabe dieser Parameter lässt sich die Szene beschreiben. Die Position eines Objekts wird folgendermaßen repräsentiert:

X - Position	Y - Position	Z – Position
10	*2*	*5*

Tabelle 1: Statische Parameter

Es soll nun die Möglichkeit geschaffen werden, die 3D-Szene auch von außen modifizieren zu können. Dazu müssen die statischen Parameter einem Platzhaltermodell weichen (im Folgenden durch die Variablen *X, Y, Z* repräsentiert). Hierzu wird ein veränderlicher Parameter *n* durch einen Platzhalter P_i wie folgt definiert: P_i mit $0 < i < n$ [vgl. Hausig98, S.7].

Dadurch ergibt sich folgendes Modell:

X - Position	Y - Position	Z – Position
X	*Y*	*Z*

Tabelle 2: Dynamische Parameter

Durch die variablen Parameter ist es möglich die Eigenschaften auch außerhalb der 3D-Szene zu beeinflussen. Diese Art der Beschreibung ist allerdings immer noch statisch. Um eine Dynamik der Parameter und damit eine zeitliche Abhängigkeit zu erreichen, muss das Modell um den globalen Parameter *t*, die Zeit erweitert werden. Daraus resultiert die Funktion: $P_i(t)$, die für jeden Zeitpunkt *t* einen Parameter P_i definiert, mit *Beginn < t < Ende*. Diese zeitliche Abhängigkeit von Parametern bildet die Basis der Keyframe-gestützten Animation.

2.3.1. Animation

Animationen basieren auf einer Täuschung des menschlichen Sehvermögens. Wird eine Serie ähnlicher, unbewegter Bilder schnell hintereinander abgespielt, so fasst das menschliche Gehirn diese als kontinuierliche Bewegung auf. Somit kann Animation als ein Trick bezeichnet werden [vgl. Oliver94, S.154]. Es existieren verschiedenste Verfahren um mit dem Computer solche bewegten Bilder zu generieren. Da im Rahmen dieser Arbeit ein Verfahren zum Einsatz kommt, welches auf der Interpolation[27] zwischen sogenannten Schlüsselbildern (nachfolgend Keyframes genannt) beruht, wird dieses Verfahren im Folgenden näher erläutert.

Keyframe-Animation
Der Begriff „Keyframe" wurde von den Disney Studios geprägt. Er bezeichnet bei Zeichentrickanimationen ein Einzelbild, welches von einem Hauptanimateur gezeichnet wird, um Position und Zeit zu bestimmen. Der Hauptanimateur ist für

[27] Mathematische Berechnung.

die Schlüsselszenen des Films zuständig. Weitere Animateure zeichnen die Bilder zwischen den Keyframes, die In-between Frames oder Tweenings [vgl. DMA07].

Der Keyframe-gestützten Computeranimation liegen Verfahren der Einzelbildanimation, sowie Produktionstechniken der analogen Filmproduktion zu Grunde. Um die Arbeit der Animatoren zu erleichtern, wurde ein Verfahren entwickelt, bei dem nicht mehr jedes Einzelbild von Hand manipuliert werden muss [vgl. Watt02A, S.525]. Bei der Keyframe gestützten Computeranimation werden, anstatt für jedes t ein explizites P_i festzulegen ebenfalls Keyframes erzeugt (Kapitel 2.3.) [vgl. Watt92, S.345]. Diese Keyframes werden dabei auf der X-Achse eines Zeitstrahls angeordnet.

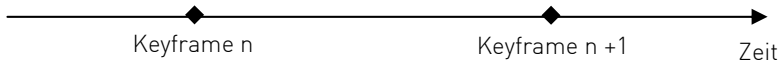


Abb. 15: Zeitlich angeordnete Keyframes

Die Zuweisung von Werten erfolgt dann über die Positionierung der Keyframes auf der Y-Achse.

Abb. 16: Wertzuweisung durch Y-Verschiebung

Die Werte, die zwischen den einzelnen Keyframes liegen werden interpoliert. keyframebasierte Systeme bieten dem Benutzer eine grafische Repräsentation des Werteverlaufs an, welche direkt manipuliert werden kann. Das geschieht in der Regel durch mathematische Kurven unterschiedlicher Stetigkeiten. Die Anzahl benötigter Keyframes hängt dabei stark von der zu erstellenden Bewegung bzw. Werteänderung ab. Normalerweise benötigt ein linearer Werteverlauf weniger Keyframes als eine komplexe weiche Bewegung.

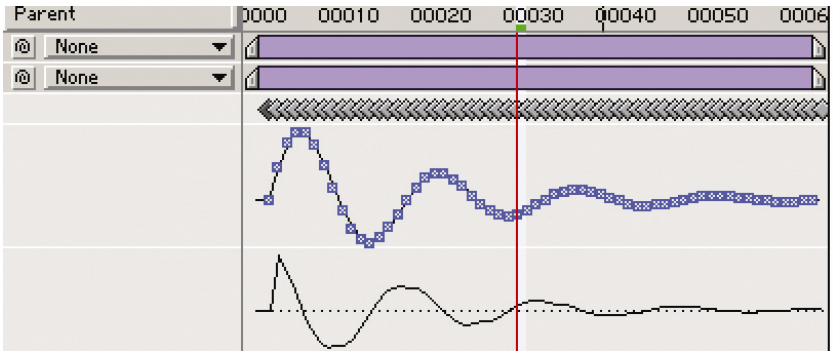


Abb. 17: Visuelle Repräsentation der Keyframe Interpolation in After Effects

Zeitleistensysteme auf Basis von Keyframe Interpolation sind weit verbreitet und bilden die Grundlage heutiger Computeranimation. Doch nicht nur dort, sondern auch in der computergestützten Musikproduktion sind keyframebasierte Systeme heute Standard.

2.4. Endliche Automaten

Ein endlicher Automat ist ein theoretisches Verhaltensmodell, welches aus Zuständen, Zustandsübergängen und Aktionen besteht. Ein Automat wird dabei als endlich bezeichnet, wenn die Menge seiner Zustände endlich ist [vgl. CL07]. Ein einfaches Beispiel eines endlichen Automaten zeigt die Abbildung 18.

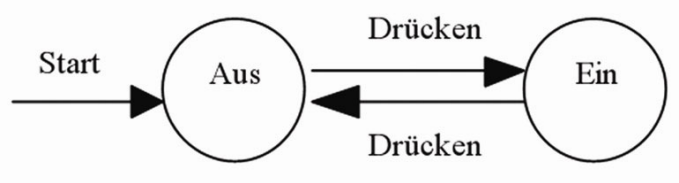


Abb. 18: Simpler endlicher Automat [vgl. Mutafchief07, S.1]

Die Abbildung zeigt das Modell eines Schalters, welcher zwei Zustände (*Ein* und *Aus*) besitzt. Als Ausgangszustand wird dabei der Zustand *Aus* definiert. Dieser Automat weiß immer in welchem Zustand er sich gerade befindet. Durch Betätigen des Schalters wird er vom Ausgangszustand *Aus* in den Zustand *Ein* versetzt und umgekehrt [vgl. Mutafchiev07, S.1].

3. Analyse

3.1. Zeitabhängige Parameterbearbeitung in VVVV

Viele der mit VVVV entwickelten Installationen sind konzeptionell darauf angewiesen zeitabhängig Parameter zu verändern. Das kann die einfache Bewegung eines grafischen Objektes auf dem Bildschirm sein, die Ansteuerung von Motoren (z.B. zur Bewegung eines realen Objektes) oder die Steuerung z.B. des Lichtes in einem realen Raum. In VVVV existieren bereits Konzepte die eine zeitgesteuerte Parameterbearbeitung erlauben. Die Vor- und Nachteile dieser Konzepte werden im Folgenden beispielhaft erläutert.

3.1.1. Knoten

VVVV bietet dem Programmierer verschiedene Knoten, die in der Lage sind Zeit zu verarbeiten. Zu den wichtigsten zählen:

- *Stopwatch*
  Knoten der einen Timer[28] startet, sobald er aktiviert ist.
- *LFO*
  In einem spezifischen, zeitlichen Intervall wird linear zwischen 0 und 1 interpoliert.
- *Damper, Oscillator, LinearFilter, DeNiro, Newton*
  Diese Knoten gehören zu der Gruppe der Filter. Sie interpolieren Werte innerhalb eines bestimmten Zeitabschnitts auf Basis verschiedener mathematischer Verfahren. So bildet der Knoten *Newton* z.B. das physikalische Prinzip der Beschleunigung eines Objekts ab.
- *FrameDelay*
  Wie in Kapitel 2.2.8. erläutert, ist eine Schleifenbildung in VVVV nicht ohne zusätzliches Konzept möglich. Der Knoten *FrameDelay* verzögert die Weitergabe eines angelegten Wertes, die Verzögerungsdauer beträgt eine

[28] Ein Timer ist ein Objekt, welches die vergangene Zeit seit seiner Aktivierung ausgibt

CPU Zeiteinheit. Durch den Knoten ist es möglich, die aus der textuellen Programmierung bekannten Schleifen (*For*, *While*) in VVVV abzubilden. So kann der Output-Pin eines Knotens mittels *FrameDelay* wiederum mit seinen eigenen Input Pins verbunden werden [vgl. VVVV07A].

Anzumerken ist jedoch, dass der exakte zeitliche Verlauf der Filterknoten selbst für geübte Nutzer sehr schwer vorhersehbar ist. Eine Zeitsteuerung auf Basis dieser Knoten wird deshalb für den Benutzer nur schwierig zu kalkulieren sein.

Um anschließend zu einem konkreten Zeitpunkt eine Aktion ausführen zu können, müssen die ausgegebenen Zeiten der vorgestellten Knoten zusätzlich mit einem oder mehreren weiteren Knoten die Bedingungen[29] abbilden (z.B. *==*, *<=*, *And*, *Or*), verknüpft werden. Danach ist es notwendig, sollte eine Bedingung zutreffen oder nicht, eine gewünschte Aktion auszuführen.

Hilfreich sind dabei die Knoten:

- *Switch*
  Mehrfachschalter, d.h. es kann zwischen zwei oder mehr Eingangswerten umgeschaltet werden.
- *Monoflop*
  Zeitschalter, Dieser Knoten schaltet nach dem Auslösen für eine angegebene Zeitspanne von 0 auf 1, danach kehrt er in den Ursprungszustand zurück.
- *FlipFlop*
  Dauerschalter, Dieser Knoten schaltet nach dem Auslösen von dem Zustand 0 in den Zustand 1. Dieser Zustand bleibt aktiv bis er durch ein weiteres Auslösen in den Ausgangszustand zurückgesetzt wird. Ein FlipFlop ist ein aus der Elektronik stammendes Konzept eines Schalters zur Speicherung eines Bit.

[29] Bedingungen geben in VVVV immer einen Booleschen Wert (0,1) zurück.

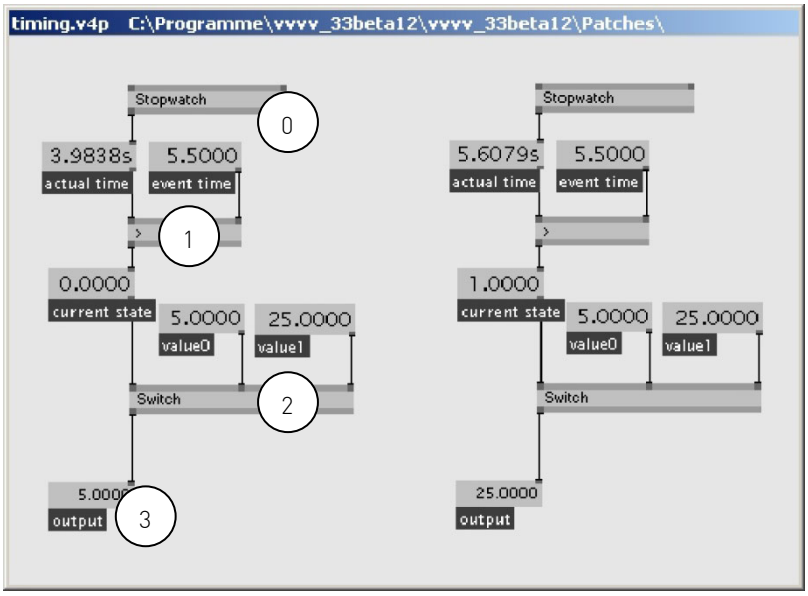


Abb. 19: Patch der eine Aktion ab einem spezifischen Zeitpunkt ausführt

Der Patch in Abbildung 19 visualisiert den exemplarischen Aufbau eines zeitgesteuerten Ablaufs. Ist ein bestimmter Zeitpunkt überschritten, so soll zwischen zwei vorgegebenen Werten umgeschaltet werden. Der Knoten *Stopwatch*(0) fungiert dabei als zeitgenerierendes Objekt. Der Knoten *grösser*(1) bildet eine Bedingung ab. So wird verglichen, ob die Zeit, welche der Knoten *Stopwatch* ausgibt einen bestimmten Wert (in diesem Beispiel *5,5*) überschritten hat. Auf der linken Seite ist die Bedingung nicht erfüllt. Der Knoten *grösser* gibt deshalb eine 0 aus. Auf der rechten Seite ist die Bedingung erfüllt. Der Knoten *Switch*(2) schaltet, sobald sein linker Input-Pin 1 ist, zwischen den beiden anliegenden Werten um.

3.1.1.1. Automation

Im vorhergehenden Abschnitt wurde gezeigt, wie es möglich ist, in VVVV Parameter zeitabhängig zu steuern. Sollte die Komplexität der gestellten Aufgabe jedoch zunehmen, sieht sich der Benutzer bald einer Vielzahl von Knoten gegenüber gestellt. Früher oder später wird er bei der vorgestellten Methode den Überblick verlieren. Eine Alternative zu den gezeigten Verfahren bietet der Knoten *Automata*. Dieser Knoten bildet einen deterministischen, endlichen Automaten ab. Er beruht dabei auf dem in Kapitel 2.4. geschilderten Prinzip. Der Knoten *Automata* bietet, entgegen der grafischen Metapher von VVVV, die Möglichkeit einer textuellen Eingabe. So kann er mit einer stark abstrahierten Form einer Programmiersprache versehen werden. Die Funktion des Knoten ist es, logische Komplexität von Verzweigungen innerhalb eines Knotens zu verbergen [vgl. VVVV07B].

Exemplarisch soll folgender Pseudo-Code einer textuellen Programmiersprache funktional in VVVV umgesetzt werden:

```
While(true)  {

    if (currentstate=reset) and (input=OnSet) then (currentstate:=set)
and (DoSet)

   if (currentstate=set) and (input=OnReset) then (currentstate:=reset)
and (DoReset)

}
```

Listing 3: Pseudo Code

Mit Hilfe des Knotens *Automata* kann die Logik aus Listing 3 wie folgt umgesetzt werden:

```
reset OnSet set DoSet

set   OnReset reset DoReset
```

Listing 4: Quadrupel Schreibweise des *Automata*-Knoten

Ergebnis dieser zwei Quadrupel[30] in Listing 4 ist eine FlipFlop Funktionalität (siehe Kapitel 3.1.1.), welche durch den *Automata* abgebildet wird.

Damit steht ein elegantes Mittel bereit um Komplexität zu verbergen. Leider wird dieser Knoten nur sehr selten eingesetzt. Aus mehreren Dialogen mit professionellen Nutzern von VVVV ging hervor, dass das Hauptproblem dieses Knoten seine fehlende Visualisierung des zu Grunde liegenden Konzeptes ist. Da der Knoten rein textuell arbeitet, wird seine Funktionalität nicht sichtbar. Mit einfachen Worten: Viele Nutzer verstehen nicht wie der Knoten funktioniert. Ein Großteil der Benutzer von VVVV kommt eher aus gestalterischen Bereichen, daher fällt es ihnen schwerer das programmiertechnische Prinzip eines endlichen Automaten zu verinnerlichen. Ein weiteres Defizit des *Automata* ist natürlich der Bruch dem grafischen Ansatz von VVVV.

[30] Ein Quadrupel bezeichnet eine geordnete Menge von vier Objekten.

3.1.2. Interface Elemente

VVVV bietet auch die Möglichkeit, zeitliche Abläufe bzw. Parameteränderungen zu visualisieren und zu editieren.

IO-Box
Die zentrale Rolle zur Datenvisualisierung bzw. Parametermanipulation nimmt in VVVV die IO-Box (Input / Output –Box) ein. Dieser Knoten ermöglicht es, durch verschiedene Modi Werte ein- bzw. auszugeben. Nach den Spezifikationen in Kapitel 2.1.5. übernimmt die IO-Box dabei sowohl die Rolle einer Datenquelle, als auch die einer Datensenke. Die wichtigsten Modi der IO-Box werden hier dargestellt. [vgl. VVVV07C]

Abb. 20: IO-Box, links mit einer-, rechts mit drei Spalten

Im einfachen Modus, in Abbildung 20 links zu sehen, kann die IO-Box sowohl Werte, die an ihren Input-Pin angelegt sind anzeigen, als auch selbst Werte generieren. Dazu lässt sich mit der Kombination der rechten Maustaste und der Bewegung der Maus auf der Y-Achse der angezeigte Wert entweder verkleinern oder vergrößern. Das in Kapitel 2.2.2. vorgestellte Inspektor-Menü gewährt jedoch wesentlich mehr Möglichkeiten zur Manipulation der IO-Box. In Abbildung 20 rechts ist zu sehen, dass sich z.B. auch mehrere Werte in Tabellenform ausgeben bzw. auch einzeln modifizieren lassen.

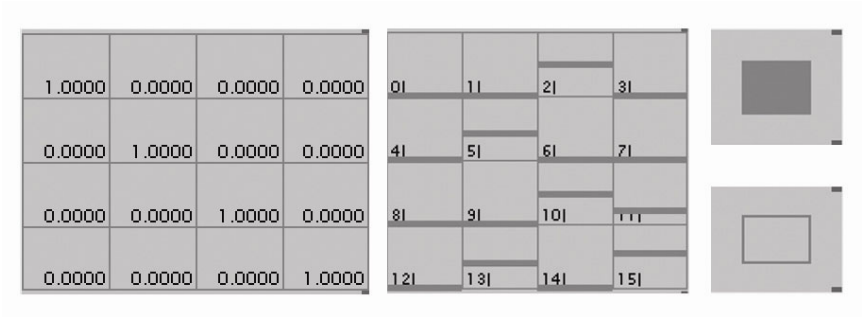


Abb. 21: Verschiedene Modi der IO-Box

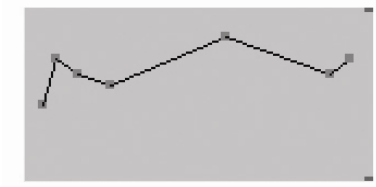


Abb. 22: IO-Box mit Linienzug

Abbildung 21 zeigt, dass die IO-Box nicht nur in der Lage ist Werte als Dezimalzahlen auszugeben, sondern diese auch als Schieberegler oder Buttons darstellen kann. Eine weitere Form ist die Darstellung von Linienzügen (Abb. 22). Allerdings können neue Stützpunkte nicht direkt hinzugefügt werden. Diese müssen einzeln über das Inspektormenü erzeugt werden.

Mit Hilfe der vorgestellten IO-Box können Interfaceelemente innerhalb des Datenstroms von VVVV realisiert werden. Theoretisch lassen sich so auch zeitabhängige Abläufe visualisieren. Allerdings bedeutet das einen erhöhten Aufwand in der Programmierung. Zum anderen ist die Genauigkeit der visuellen Darstellung von Werten mittels der IO-Box nicht besonders hoch, da weder Maßeinheit noch andere Hilfsmittel, wie z.B. eine Skala, existieren. Das hat meist ein Abschätzen des Angegebenen Wertes zur Folge. Die Abbildung 23 zeigt die Visualisierung mehrerer zeitlicher Abläufe. Sie zeigt weiterhin, dass auch der visuelle Platzverbrauch nicht zu unterschätzen ist.

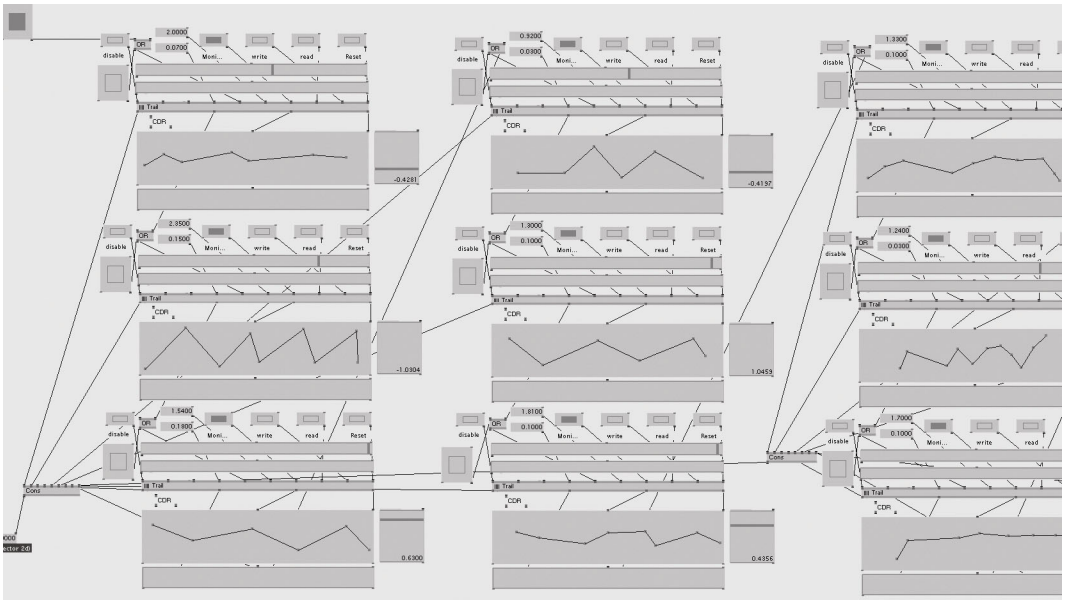


Abb. 23: Visulisierung zeitlicher Abläufe mit Hilfe der IO-Box

Direct-X

Eine andere Variante der Interfacegestaltung für zeitliche Abläufe wäre die Konstruktion eigener Benutzerschnittstellen unter Benutzung der in VVVV integrierten Direct-X Renderer. Da allerdings keine Knoten existieren, die eigenständige Interfaceelemente auf Direct-X Basis anbieten, müssten alle Komponenten, z.B. Buttons, mit VVVV selbst erstellt werden. Diese Möglichkeit stellt sich als derart komplex dar, dass sie im Rahmen dieser Arbeit nicht näher behandelt werden kann. Es kann aber festgestellt werden, dass dieser Lösungsansatz mit einer immensen Arbeits- und Ressourcenaufwendung verbunden wäre.

3.1.3. Anwendungsfall

Im Folgenden wird kurz ein Anwendungsfall untersucht. Diesem liegt eine zeitlich basierte Parametersteuerung, auf Basis der zuvor beschriebenen Elemente zu Grunde. Anhand dieser wird exemplarisch verdeutlicht, worin die Probleme des aktuellen Konzeptes liegen.

Ziel der Medieninstallation „Das apokalyptische Weib“ war es, ein komplettes Kirchenschiff auszuleuchten, sowie auf der zylinderförmigen Decke eine zeitgesteuerte Animation zu zeigen. Die Licht- und Raumverhältnisse ließen sich auf Grund der Ausmaße des Bauwerks nur sehr schwer und ungenau simulieren. Deshalb galt es ein System zu entwerfen, welches das genaue Justieren der gewünschten Parameter, wie z.B. die Entzerrung der Projektion, die Feineinstellung von Ablaufzeiten oder die Farbgebung vor Ort ermöglichte. VVV war dabei Basis für die Realisierung. Anhand dieses Beispiels wird dargestellt, worin die Problematik zeitgesteuerter Abläufe in VVV liegt.

Zeitgeber

Bei der Gestaltung der Animation war es notwendig, zwischen zwei globalen, unterschiedlich langen Szenen zu unterscheiden. Nach Ablauf der ersten Szene sollte automatisch die zweite Szene beginnen, danach wieder zur ersten gesprungen werden. Die Abbildung 24 zeigt den Patch, der diese Funktionalität realisiert.

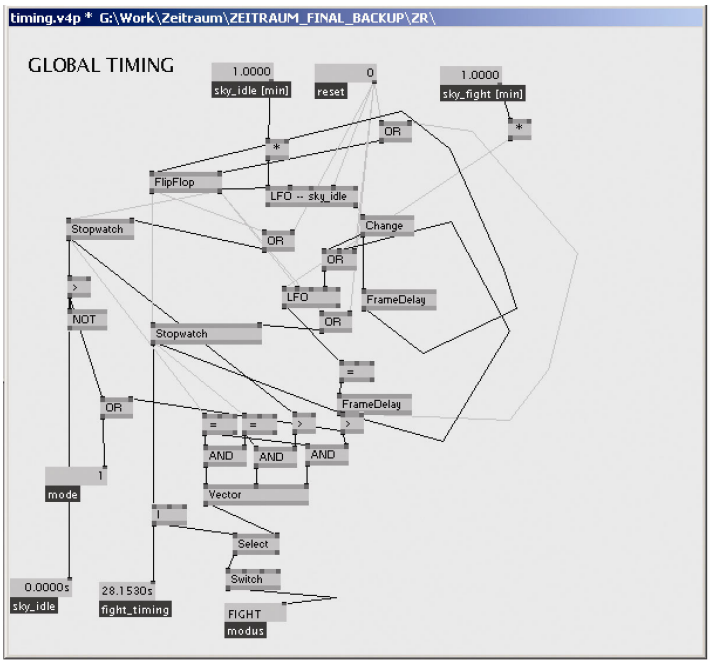


Abb. 24: Zwei Stopuhren die nacheinander ablaufen

In diesem Patch werden verschiedene der in Kapitel 3.1.1. vorgestellten Knoten verwendet. Die Komplexität ist zumindest für einen geübten Nutzer beherrschbar. Allerdings ist der Umfang des logischen Aufbaus im Vergleich mit dem erzielten Ergebnis bereits relativ hoch.

Zeit-Manager

Deutlicher wird dieser „Aufwand / Nutzen“ Effekt am Beispiel des Zeitmanagers. Um den eigentlichen zeitgesteuerten Ablauf der Animation und des Tons zu steuern wurde ein Patch erstellt, der an bestimmten Zeitpunkten Aktionen mit bestimmter Dauer ausführt. Es handelt sich dabei z.B. um Überblendungen von Farben, die Steuerung von verschiedenen visuellen Effekten innerhalb von Pixel

und Vertexshadern, oder auch um das Abspielen von Ton. Abbildung 25 zeigt einen Ausschnitt des Patches.

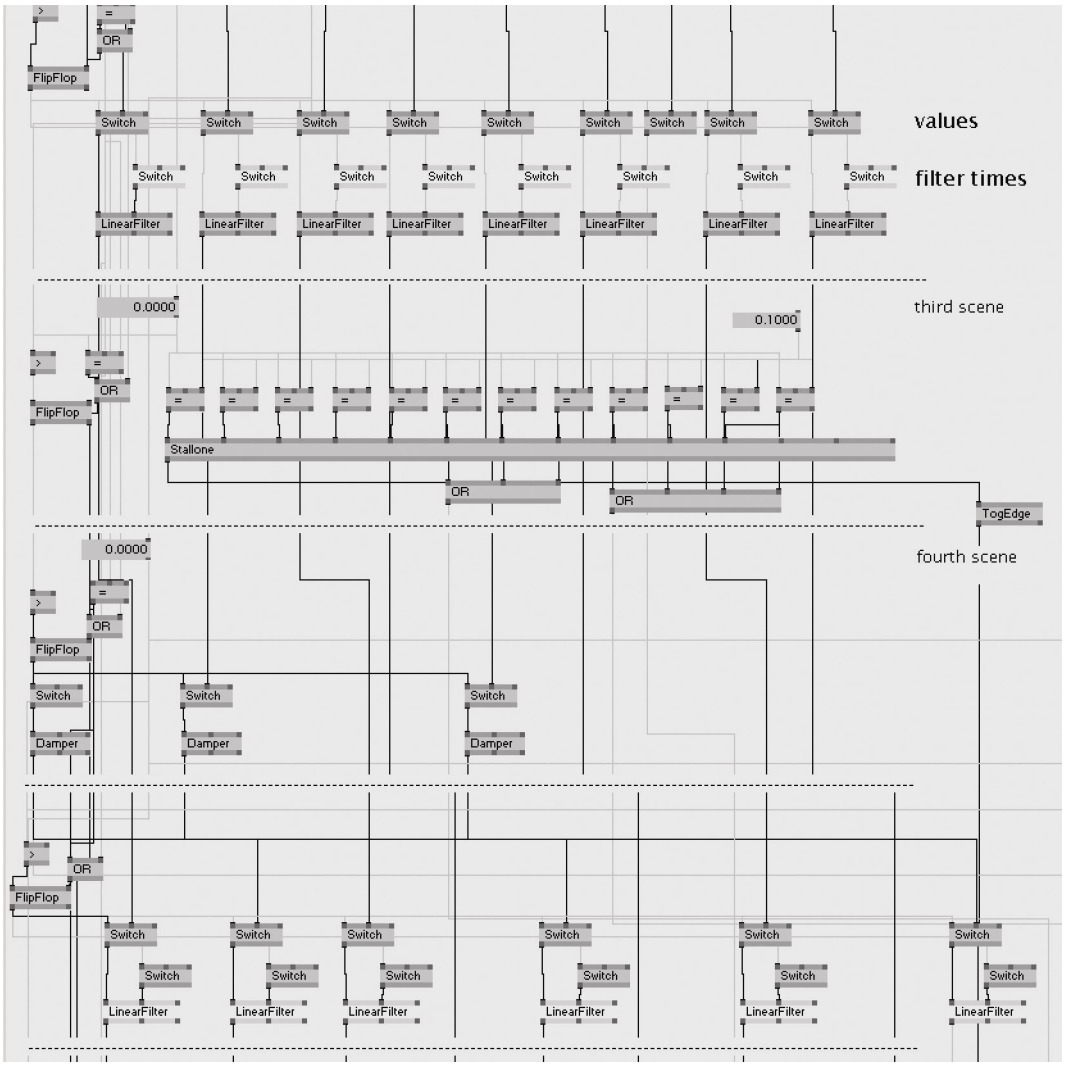


Abb. 25: Zeitbasierte Parametersteuerung in VVV

Wie das Bild zeigt, ist dieser Patch trotz der Vorteile der visuellen Programmierung sehr unübersichtlich. Die Funktionalität ist selbst für einen geübten Nutzer kaum nachvollziehbar. Auch die Verknüpfung der auszuführenden Aktionen ist sehr umständlich und unübersichtlich. Bedarf es beispielsweise einer partiellen Änderung im zeitlichen Ablauf, müssen zusätzlich alle nachfolgenden Werte manuell angepasst werden. Von „Editieren" kann also keine Rede sein. Ein weiterer großer Nachteil ist, dass sich der zeitliche Ablauf nicht visuell erfassen lässt. Die hier vorhanden zeitlichen Parameter liegen nur in Form von Dezimalzahlen vor. Es ist somit ein sehr abstraktes Modell Parameterverläufe zu realisieren.

3.1.4. Ressourcen

Ein weiterer nicht zu vernachlässigender Aspekt, ist der Verbrauch von Ressourcen. So muss derzeit für jedes Projekt eine individuelle Zeitsteuerung angefertigt werden. Da viele Projekte die Meso für Kunden realisiert in einem relativ kurzen Zeitfenster umgesetzt werden müssen, würde eine fest integrierte Lösung von sehr hohem Nutzen sein. Dies würde es ermöglichen wesentlich mehr Zeit für die eigentlich zu lösende Aufgabe aufzuwenden, finanzielle Mittel einzusparen und die Nutzungsdauer der Hardware zu verkürzen.

Eine andere Art von Ressource stellt der visuelle Raum auf dem Bildschirm dar. Dieses ist ein generelles Problem der visuellen Programmierung, welches durch die derzeitige Lösung in VVVV noch künstlich verstärkt wird. Der Knoten *Automata* bietet hier zwar Abhilfe, jedoch wird er auf Grund der aufgezeigten Nachteile nur selten eingesetzt.

3.1.5. Zwischenfazit

Wie in den letzten Abschnitten deutlich wurde, ist es in VVVV zwar möglich, zeitgesteuert Parameter zu bearbeiten, aber nur mit starken Einschränkungen. Die gezeigten Lösungsansätze bleiben nur fortgeschrittenen Nutzern vorbehalten und sind oft nur für den Benutzer nachvollziehbar, der diese auch erstellt hat. Auch der Zeitverbrauch einer zu erstellenden Lösung ist enorm. Der Grad an Visualisierung ist sehr gering. Auch durch die Verwendung von IO-Boxen kann das nur bedingt verbessert werden. Selbst die Nutzung der Direct-X Komponenten bringt hier keine Verbesserung, da im Moment keine Komponenten vorhanden sind, die eine effiziente Erstellung eines Benutzerinterface erlauben würden.

3.2. Keyframe Interpolation

Wie in Kapitel 2.3.1. bereits kurz angedeutet, basieren Systeme die den Keyframe-Ansatz verfolgen auf der Interpolation der Werte zwischen den Keyframes. Die Basis dieser Interpolationen sind verschiedene mathematische Verfahren.

3.2.1. Lineare Interpolation

Die einfachste und am häufigsten benutzte Methode der Interpolation zwischen zwei Keyframes bildet die lineare Interpolation. Sie benötigt mindestens zwei Keyframes mit unterschiedlichen Werten. Bei der linearen Interpolation ändern sich die Werte konstant. Sie bildet eine Gerade zwischen zwei Punkten ab.

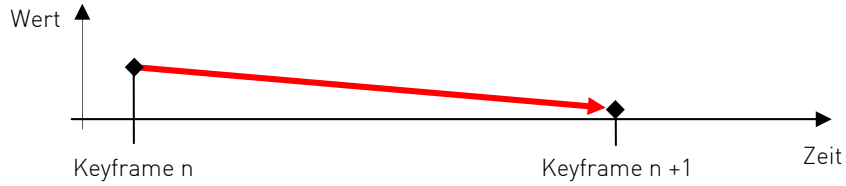


Abb. 26: Lineare Interpolation

3.2.2. Höhergradige Polynome

Meist offerieren Animationssysteme auch höhergradige Polynome zur Interpolation zwischen zwei Keyframes. Diese bieten, im Gegensatz zur linearen Interpolation, einen weichen Kurvenverlauf und sind daher für Bewegungsanimationen besser geeignet. Der Interpolation eines Kurvenverlaufs liegen dabei Kontrollpunkte zu Grunde. Mit ihnen kann das Verhalten einer Kurve auf unterschiedliche Weise beeinflusst werden. Da mehrere Verfahren zur Abbildung eines Kurvenverlaufs existieren, werden im Folgenden die wichtigsten aufgeführt. In der Computergrafik werden zumeist nur Polynome des dritten Grades verwendet, da ein höherer Grad eine aufwendige Berechnung nach sich zieht und die Kurve mit steigendem Grad zu schwingen beginnt. Ein Polynom quadratischen Grades ist aber in vielen Fällen nicht flexibel genug, weshalb kubische Polynome verwendet werden. Der Grad eines Polynoms ist dabei immer um eins niedriger als die Anzahl seiner Kontrollpunkte [vgl. Watt02, S. 94].

3.2.2.1. Bézier Interpolation

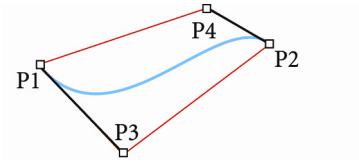


Abb. 27: Durch Kontrollpunkte aufgespannte konvexe Hülle einer Bézierkurve

Eines der bekanntesten Verfahren zur Beschreibung einer Kurve in der Computergrafik ist die Bézier-Interpolation. Eine Bézierkurve des dritten Grades besitzt vier Kontrollpunkte. Der Kurvenverlauf ist dabei durch eine sogenannte konvexe Hülle bestimmt - ein Polygon, welches durch die Kontrollpunkte aufgespannt wird (Abb. 27). Bézierkurven besitzen einen vorhersehbaren Kurvenverlauf. Jeder Punkt auf der Kurve wird durch die kubische Skalierung jedes Kontrollpunkts definiert. Dies wird als Basisfunktion bezeichnet [vgl. Watt02, S. 94].

Der erste und letzte Kontrollpunkt bilden jeweils Start- und Endpunkt der Kurve. Wird ein Kontrollpunkt zwischen Start- und Endpunkt bewegt, so wird die Kurve in der Nähe des Punktes gestaucht oder gestreckt. Je grösser der Abstand vom manipulierten Kontrollpunkt, desto geringer die Auswirkung auf die Veränderung

der Kurve. Die Kontrollpunkte können daher als eine Art Gewicht verstanden werden [vgl. Günther01, S. 5].

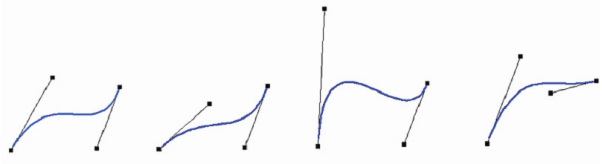


Abb. 28: Beeinflussung des Kurvenverlaufs durch die Kontrollpunkte [vgl. Günther01, S. 5]

Bei der Bézier-Interpolation wirkt sich das Verschieben eines Kontrollpunkts mehr oder weniger (je nach Entfernung) auf die gesamte Kurve aus. Die Modifikation ist nicht lokal begrenzt [vgl. Watt02, S.94]. Die gesamte Kurve kann transformiert werden, indem beliebige affine Transformationen[31] auf ihre Kontrollpunkte angewendet werden. Dabei ist die Kurve invariant, d.h. ihre Form wird nicht verändert [vgl. Watt02, S.94].

Da es bei der Keyframe-Interpolation mehr als nur eines Kurvenabschnitts bedarf muss überlegt werden, wie eine längere Kurve definiert wird. Dazu werden mehrere Kurvenelemente zusammengesetzt. Das Ergebnis wird als stückweise Polynomkurve bezeichnet [vgl. Watt02, S.92]. Alternativ kann auch der Grad der Kurve erhöht werden. Von diesem Vorgang wird aber, wegen den zuvor beschriebenen Nachteilen, abgesehen. Die Verbindung von zwei oder mehr Kurvenelementen erfordert, dass Bedingungen für den Anschlusspunkt gelten. Die einfachste der Bedingungen lässt sich definieren durch: $P0_{(i+1)} = P3_{(i)}$

Das bedeutet, dass der Endpunkt der aktuellen Kurve gleich des Startpunktes der nächsten Kurve ist und wird als Stetigkeit der Position bezeichnet [vgl. Watt02, S.92]. Aufgrund dessen lassen sich mehrere Kurven miteinander verbinden. Auch für die Kontrollpunkte müssen Regeln definiert werden. Daher werden Anschlussbedingungen tangentialer Stetigkeit der ersten Ordnung benötigt. Die Kanten des charakteristischen Polygons sind kollinear und die Tangentenvektoren am Ende der einen Kurve und am Beginn der anderen Kurve dürfen sich nur in einer Konstante unterscheiden.

Die Stetigkeit erster Ordnung wird bewahrt durch: $(S_3 - S_2) = k(R_1 - R_0)$

Wobei S_i und R_i die Kontrollpunkte der beiden Kurvenabschnitte repräsentieren. Abbildung 29 verdeutlicht die verschiedenen Bedingungen.

[31] Linearkombination linearer Transformationen

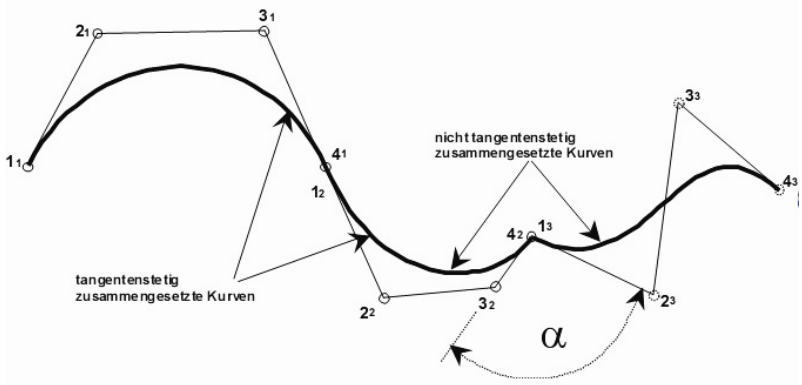


Abb. 29: Drei verbundene Bézierkurven. Links mit tangentialer Stetigkeit, rechts mit Positionsstetigkeit [vgl. Noack, n.A.]

3.2.2.2. Kochanek-Bartels Splines

Eine weitere Form der Interpolation, welche häufig in Animationssystemen Anwendung findet, ist die Interpolation mit Hilfen von kubischen Kochanek-Bartels Splines. Ein Kochanek-Bartels Spline ist eine Erweiterung eines Catmull-Rom Splines, welcher eine Ableitung eines kardinalen Splines ist [vgl. Cubic07]. Kardinale Splines sind wiederum Subtypen der Hermite Interpolation. Diese beschreibt einen kubischen Kurvenverlauf an Hand von Start –und Endpunkt und zwei Tangentenvektoren [vgl. Armstrong05, S.1]. Kardinale Splines hingegen besitzen keine Tangentenpunkte. Diese werden algorithmisch aus den Kontrollpunkten berechnet. Die Berechnung der Tangentenpunkte erfolgt mittels folgender Gleichung: $T_i = a(P_{(i+1)} - P_{(i-1)})$ [vgl. Pipenbrink98]

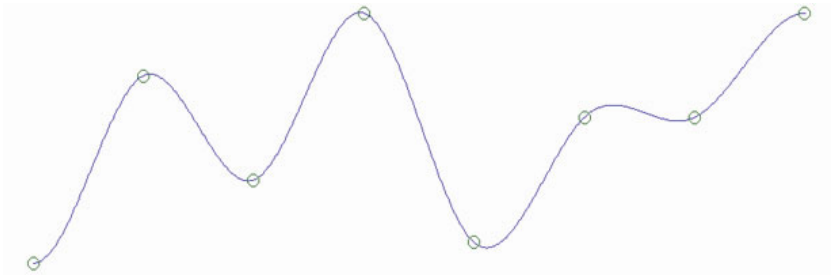


Abb. 30: Spline mit berechneten Tangenten

Wobei *a* die Steigung der Kurve angibt und in einem Intervall [0,1] liegen sollte [vgl. Cubic07]. Catmull-Rom Splines setzen den Parameter a als Konstante auf den Wert 0.5. Daraus ergibt sich folgende Gleichung: $T_i = 0.5(P_{(i+1)} - P_{(i-1)})$ [vgl. Thalmann, n.A., S.6]

Somit ist es möglich die Kurve direkt durch die Kontrollpunkte laufen zu lassen, ohne zusätzlich Tangentenpunkte nutzen zu müssen. Die Variante der Kochanek-Bartels Splines fügen dem Catmull-Rom Spline nun Parameter hinzu, mit denen es möglich ist, die Steigung der Kurve lokal zu kontrollieren.

Diese Parameter sind[32]:

- *Tension (Spannung)*
  Bestimmt den Krümmungsgrad der Kurve.
- *Continuity (Kontinuität)*
  Gibt den Kurvenverlauf (Geschwindigkeit) am Kontrollpunkt vor.
- *Bias (Neigung)*
  Definiert, wo die Kurve in Bezug auf den Kontrollpunkt auftrifft.

Aus diesem Grund werden sie auch TCB-Splines genannt.

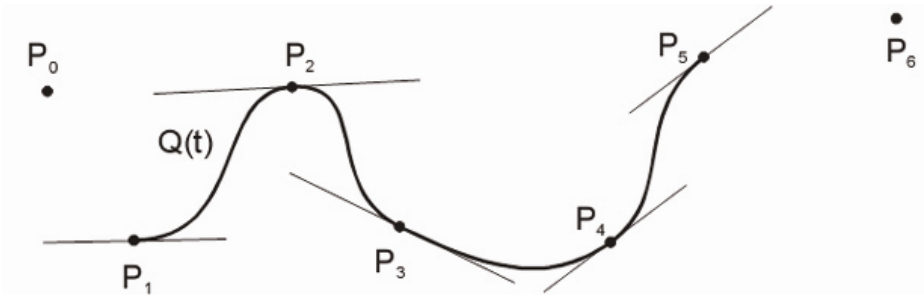


Abb. 31: TCB-Spline [vgl. Benes01, S.4]

3.3. Zeitleistenbasierte Systeme

Wie bereits in Kapitel 3.3. angesprochen, finden zeitleistenbasierte Systeme vor allem in der Computeranimation, dem Videoschnitt, sowie der Musikproduktion ihre Anwendung. Zwei solcher Werkzeuge werden hier in Bezug auf ihre Funktionsweise näher betrachtet.

Zunächst erfolgt eine Veranschaulichung des Systems 3D-Studio Max, welches mit seinem sehr umfangreichen zeitleistenbasierten Editor einen guten Eindruck des heutigen Standes moderner Animationssysteme bietet. Als Zweites wird die Zeitleistensteuerung der visuellen Programmiersprache Max untersucht, um zu sehen, wie sich die Konzepte linearer Bearbeitung mit einem Echtzeitsystem vereinbaren lassen. Ferner wird erörtert, ob und wie sich die eingesetzten Verfahren auf die Problemstellung dieser Arbeit anwenden lassen. Aufgrund der Komplexität von zeitleistenbasierten Editoren können in dieser Arbeit jedoch nur die grundlegenden und wichtigsten Funktionsweisen untersucht werden.

Vorweggreifend kann festgestellt werden, dass zeitleistenbasierten Systeme neben dem Zeitstrahl über eine Abspiellogik, sowie einen Abspielzeiger verfügen. Die Abspiellogik ist dafür verantwortlich, dass der Abspielzeiger sich in einer bestimmten Weise auf dem Zeitstrahl bewegen kann. Der Abspielzeiger gibt dann

[32] [vgl. Eberly06, S.1]

den Wert zurück, der sich aus den vorliegenden Keyframe-Daten, auf die er zeigt, errechnen lässt.

3.3.1. 3D Studio Max

Einer der bekanntesten Vertreter der Animationssysteme ist 3D Studio Max der Firma Autodesk. Das, vorranging im semiprofessionellen Bereich eingesetzte System bietet dem Benutzer, von der Modellierung einer 3D-Szene bis hin zur Generierung dynamischer Effekte, viele Möglichkeiten der Bild- oder Animationskomposition. Dabei verfügt 3D Studio Max über ein komplexes keyframebasiertes Zeitleistensystem zur Animationssteuerung, welches nachstehend untersucht wird.

In 3D Studio Max kann nahezu jeder Parameter eines virtuellen Objekts zeitlich verändert (animiert) werden [vgl. Polevoi99, S.537]. Basis für diese Parametermanipulationen ist das Track View Panel, ein zeitleistenbasierter Keyframe Editor.

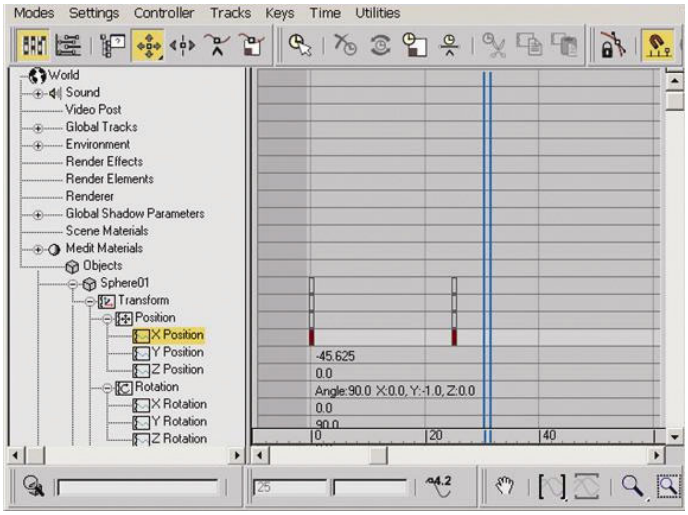


Abb. 32: Track View Panel, die Zeitleistensteuerung in 3D Studio Max

Diesem Editor können veränderliche Parameter einer 3D-Szene zugewiesen werden, z.B. indem der Benutzer in der Szene ein Objekt markiert und einen neuen Keyframe für dieses Objekt erstellt. Daraufhin wird das Objekt automatisch dem Editor hinzugefügt. Es werden Aufnahmespuren erzeugt, die jeweils einen Parameter des Objekts abbilden. Dabei werden für ein Objekt immer so viele Spuren erzeugt, wie veränderliche Parameter existieren. Spuren, die für jedes Objekt existieren, sind z.B. Rotation, Translation und Skalierung. Hinzu kommen noch viele weitere Objektspezifische Parameter. Die Spuren sind im Track View Panel (Abb. 32) hierarchisch in einer Baumstruktur angeordnet. Innerhalb einer Spur können Keyframes erstellt bzw. manipuliert werden, um die Objektparameter zu verändern. Die Basis hierfür bilden parametrische Kurven.

Dabei wird für jede der drei Dimensionen eine separate Kurve erzeugt. In den Standardeinstellungen existieren für die drei Basis Track Typen eines Objekts (Translation, Rotation, Skalierung) unterschiedliche Interpolationsmethoden; für die Translation und Skalierung wird das in Kapitel 3.2.2.1. beschriebene Bézier-Verfahren eingesetzt. Die Keyframes bilden jeweils die Endpunkte eines Kurvensegments ab. Die Interpolation kann dann mittels der Tangenten Kontrollpunkte beeinflusst werden. Für die Rotation kommen dagegen die in Kapitel 3.2.2.2. beschriebenen TCB-Splines zum Einsatz. Diese werden verwendet, um Quaternionen[33] für die Rotation eines Objekts im dreidimensionalen Raum nutzen zu können. Mit der Verwendung des Bezier-Verfahrens wäre das nicht möglich [vgl. Polevoi99, S.592]. Neben diesen beiden Verfahren existieren noch weitere, die Spur-übergreifend gewählt werden können. *Noise*, welches eine Kurve auf Grundlage eines Perlin-Noise[34] Algorithmus erzeugt, oder *Euler,* mit dem die direkte Manipulation von Eulerschen Winkeln[35] möglich wird, sind zwei dieser Möglichkeiten.

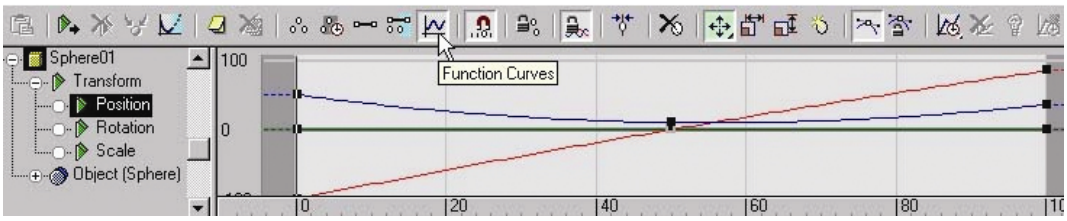


Abb. 33: Simultane Modifizierung mehrerer Kurven

Zur Interpolationskontrolle zwischen den Keyframes bietet 3D Studio Max die Möglichkeit, neben der manuellen Manipulation der Handles, verschiedene vorgefertigte Varianten von Handle-Eigenschaften auszuwählen. Die Handle-Position kann z.B. so gewählt werden, dass automatisch eine stetige Kurve, oder auch ein linearer Kurvenverlauf entsteht.

Weiterhin kann mehr als eine Kurve gleichzeitig manipuliert werden. Dazu können wahlweise die Komponenten eines Tracks zusammen betrachtet und editiert werden. Abbildung 33 zeigt die gleichzeitige Darstellung der Kurven für die 3 Achsen einer Translation. Erstellt der Benutzer in dieser Ansicht einen neuen Keyframe, so wird dieser in allen Kurven gleichzeitig generiert. In einem weiteren Modus blendet das Track View Panel alle Interpolationskurven aus und zeigt die Spuren nur noch eindimensional. Die Keyframes bleiben trotzdem sichtbar. Dadurch eröffnet sich die Möglichkeit, den zeitlichen Ablauf separat von der Wert-Parametrisierung zu editieren.

---

[33] Erweiterung des reellen Zahlenraums, besonders für Drehungen im 3D-Raum geeignet.

[34] Fraktal Algorithmus mit dem endlose Gebilde anhand weicher Steigungen errechnet werden können.

[35] Mögliche Beschreibung der Orientierung im 3D-Raum mit Hilfe von Winkeln.

3.3.2. Max

Das, bereits in der Einleitung dieser Arbeit erwähnte Max ist, wie VVVV, ein datenstromorientiertes Echtzeitprogrammiersystem. Im Gegensatz zu VVVV liegt der Fokus von Max jedoch auf der Generierung Audiokomponenten. So ist es z.B. möglich, Synthesizer virtuell zu konstruieren. Es steht dafür, wie in VVVV, eine Vielzahl von Knoten zur Verfügung.

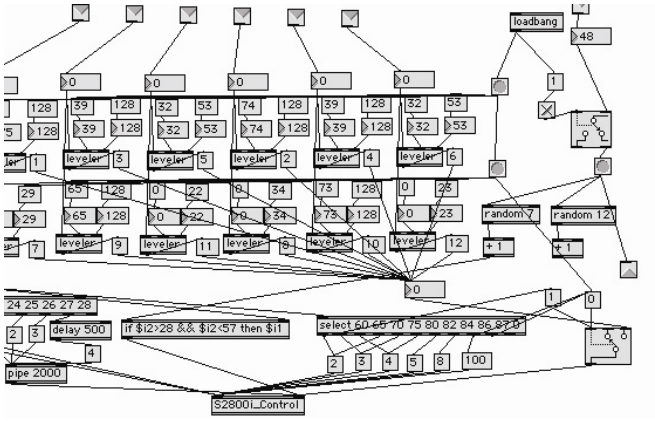


Abb. 34: Patch in Max

Durch die Knoten *Ticmd* und *Timeline* wird in Max eine Zeitleistensteuerung implementiert. Der Knoten *Timeline* übernimmt dabei die Rolle eines grafischen Editors. Der Knoten *Ticmd* ist für die Kommunikation eines Max Programms (in Max und im Folgenden „Patch" genannt) und der Zeitleistensteuerung verantwortlich. Der Knoten *Timeline* wird in Max nicht direkt in den Datenstrom integriert, d.h., er hat zwar einen Input-Pin mit dem Einstellungen der Zeitleistensteuerung vorgenommen werden können, aber keinen Output-Pin. Die Ausgabe der Werte geschieht über den Knoten *Ticmd*. Wird der Knoten *Timeline* angelegt, so öffnet sich zunächst ein Editorfenster. Nun ist es möglich, Patches, die speziell vorbereitet wurden, in den Editor zu laden. Diese Patches heißen in Max Action-Patches und werden als einzelne Spuren abgebildet [vgl. Max06, S. 612 f.]. Dabei basiert die Zeitleistensteuerung von Max nicht auf dem keyframebasierten Ansatz. Auf der Zeitleiste können sogenannte Messages arrangiert werden, die zuvor in einem Action-Patch definiert wurden. Messages entsprechen in Max Variablen ohne vorherige Typisierung. Messages können sowohl Float-Werte, aber auch Strings oder andere Datentypen sein. Eine Interpolation von Werten ist im Zeitleisteneditor von Max nur über einen Umweg möglich. So kann in Max eine grafische User Interface Komponente ebenfalls als Message betrachtet werden. Diese Komponente kann z.B. einen Linienzug repräsentieren. Wird dieser Linienzug einer Spur zugeordnet, so kann mit ihm eine lineare Interpolation zwischen Werten erfolgen. Andere Interpolationsarten bietet Max hingegen nicht. Es ist aber möglich verschiedene Message-Typen innerhalb eines Tracks zu arrangieren. Dabei spielt lediglich die Positionierung auf der X-Achse eine programmtechnische Rolle. Die Anordnung auf der Y-Achse erfüllt nur den Zweck, mehrere Messages gleichzeitig ausgeben zu können.

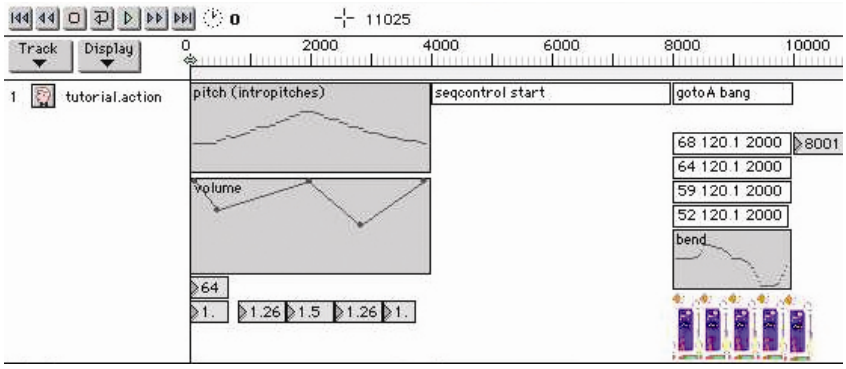


Abb. 35: Zeitleistensteuerung in Max

Die Ablaufsteuerung der Zeitleiste kann direkt durch den grafischen Editor, wie auch von außen, durch den Input-Pin des Knotens (in Max kann ein Input-Pin mehrere Werte sequentiell verarbeiten), erfolgen.

3.3.3. Zwischenfazit

Durch die Betrachtung der vorgestellten Beispielsysteme scheint eine auf mehreren Spuren angelegte Steuerung zeitlicher Parameter vorteilhaft zu sein. Die dadurch zur Verfügung gestellte Gleichzeitigkeit der Parametrisierung von Objekteigenschaften erlaubt dem Benutzer den intuitiven Umgang mit komplexen Szenarien. Bei 3D Studio Max fällt besonders die Vielzahl der Editier-Varianten für Keyframes ins Gewicht. Die Auswahl verschiedener Interpolationsmodi innerhalb einer Spur bietet, besonders bei der Animation von virtuellen Objekten, große Freiheit für den Benutzer. Weiterhin ist die Möglichkeit mehrere Kurven innerhalb einer Spur gleichzeitig Betrachten und Editieren zu können wichtig, wenn es darum geht verschiedene Parameter aufeinander abzustimmen. Allerdings wird in 3D Studio Max die Zweckbestimmung der Parametersteuerung deutlich. So wird eine Kurve immer einer expliziten Eigenschaft zugewiesen. Es ist demnach nicht möglich, z.B. die Kurve die eine Rotation beschreibt, auch einem- oder mehreren anderen Parameter zuzuordnen, sodass sie z.B. gleichzeitig Skalierung und Translation eines anderen Objektes steuert. Diese Zweckbestimmung soll bei dem zu implementierenden System, aufgehoben werden. Wie die Realisierung einer nicht-zweckbestimmten Parametrisierung erreicht werden kann, macht in diesem Fall die Konzeption der in Max implementierten Zeitleistensteuerung deutlich. Da hier die Parameterwerte lediglich als Output-Pin vorliegen und nicht explizit mit einem Objekt verknüpft sind, kann der Benutzer entscheiden, was mit den Werten parametriert wird. Dadurch wird es möglich, auch mehrere Eigenschaften verschiedener Objekte von einer einzelnen Kurve abhängig zu machen. Da auch Max ein Echtzeitsystem ist, kann die Verknüpfung einer Kurve mit einem Objektparameter zur Laufzeit aufgehoben und neu verankert werden, bzw. können während des Ausführens des Programms neue Objekte hinzugefügt und mit der Zeitleistensteuerung verbunden werden. Dies ermöglicht ein äußerst intuitives Arbeiten mit VVV im

grafischen Bereich. Max bietet zudem die Option auch andere Datentypen mit der Zeitleistensteuerung zu manipulieren. Wird der Aspekt einbezogen, dass VVVV oft für die Realisierung von grafischen Benutzerschnittstellen verwendet wird, so eröffnen sich sicherlich viele neue Möglichkeiten. Es ist denkbar, dass ein Benutzerinterface die Textinhalte, z.B. für Mehrsprachigkeit, direkt aus der Zeitleiste erhält. Leider wurde die Integration der Zeitleistensteuerung in den Datengraph von Max umständlich gelöst. Ein zweiter Knoten ist daher erforderlich um die Daten in einem Patch verwerten zu können. Des Weiteren müssen bereits im Vorfeld alle Parameter eines Patches festgelegt werden, um sie in der Zeitleistensteuerung nutzen zu können. Die Steuerung der Zeitleiste durch einen Patch ist ein sehr wichtiger Punkt. Nur ist der Parameterzugriff in Max ziemlich umständlich und unübersichtlich, da alle Messages über einen einzigen Input- oder Output-Pin übergeben werden. Der Nutzer muss sich selbst darum kümmern gewünschte Daten zu extrahieren. Ein letzter interessanter Aspekt der Zeitleistensteuerung von Max, ist das Prinzip der sogenannten Bangs. Ein Bang ist in Max eine Message, die einem Booleschen Wert entspricht. Ein Bang wird genutzt um z.B. Aktionen auszuführen. Der zuvor beschriebene Knoten *Ticmd* besitzt einen Output-Pin, welcher immer dann einen Bang sendet, wenn der Abspielknopf eine Message auf der Zeitleiste erreicht. Im Programmierkonzept von VVVV existiert ein ähnliches Verhalten. Allerdings ist dieses nicht fest integriert, sondern muss vom Benutzer durch Knoten implementiert werden (z.B. der Knoten Change). Es wäre wünschenswert eine ähnliche Funktionalität für das zu entwickelnde System nutzen zu können.

3.4. Schnittstelle zu VVVV

Bevor auf die expliziten Anforderungen des zu entwerfenden Systems eingegangen wird, findet zunächst eine Untersuchung darüber statt, wie sich eine Zeitleistensteuerung in die Programmierumgebung VVVV integrieren lässt. Da verschiedene Möglichkeiten existieren eine Implementierung zu vollziehen, werden diese im Folgenden kurz diskutiert.

Virtueller Knoten
Eine erste denkbare Form der Implementierung wäre eine vollständige Realisierung der Zeitleistensteuerung mittels VVVV selbst. Aus diesem Grund kann überlegt werden, das Konzept des Subpatching einzusetzen. Subpatching erlaubt es, innerhalb von VVVV Komplexität zu verbergen. Subpatches sind virtuelle Knoten, d.h. sie sind nicht fest im Quellcode verankert, sondern werden innerhalb von VVVV selbst, durch andere Knoten, erzeugt. Ein Subpatch ist demnach eine Subroutine, ein Patch im Patch. Dabei wird er dem Benutzer als Knoten symbolisiert. In einem Subpatch können mit Hilfe der in Kapitel 3.1.2. erläuterten IO-Box Datenquellen sowie Datensenken definiert werden. Diese werden am virtuellen Knoten als In- und Output-Pins visualisiert.

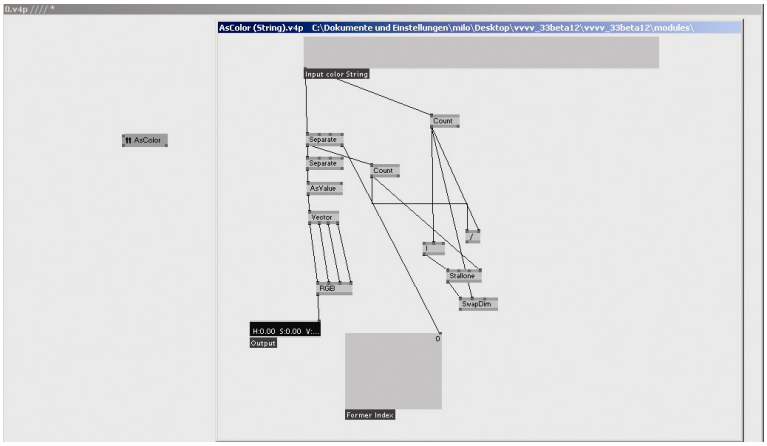


Abb. 36: Subpatch in VVVV

Mit Hilfe der in Kapitel 3.1.1. vorgestellten Verfahren wäre damit eine modulare Zeitleistensteuerung durch VVVV selbst möglich.

Dedizierter Knoten

Alternativ könnte der Knoten direkt in den Quellcode von VVVV integriert werden. VVVV ist in Delphi programmiert; dadurch würden sich alle Vorzüge einer objektorientierten Entwicklung offerieren. Ferner bietet dies die Möglichkeit, durch die, in Delphi vorhandenen Komponenten, ein grafisches Interface in einem separaten Editorfenster zu entwickeln.

Externer Knoten

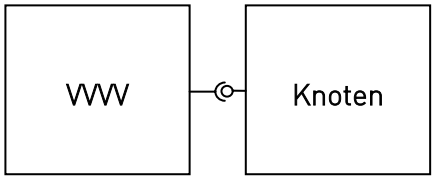


Abb. 37: Interface zu VVVV

Der dritte Weg einer Implementierung wäre die Einbindung als Knoten über eine Plug-In Schnittstelle. Es ist vorstellbar, ein Interface zu nutzen, dass alle Möglichkeiten einer direkten Implementierung bietet, dabei aber nicht die direkte Integration in den VVVV Quellcode erfordert und so auch die Wahl einer anderen Programmiersprache gestattet.

4. Anforderungen

Im Folgenden werden die Anforderungen an ein System zur zeitgesteuerten Parametersteuerung innerhalb von VWV definiert. Im Kapitel 3 wurde der aktuelle Zustand in VWV analysiert und Beispielsysteme untersucht. Es gilt nun eine klare Zielstellung zu formulieren.

4.1. Zielsetzung

Es soll untersucht werden, inwieweit sich der Echtzeitansatz von VWV mit dem Konzept einer Zeitleistensteuerung vereinbaren lässt und welche neuen Bearbeitungsmethoden sich daraus ergeben. Vordergründig ist dabei zu beachten, dass das zu entwickelnde System keinerlei Zweckbestimmung unterliegen soll. Das wird deutlich, wenn berücksichtigt wird, dass keine Integration in einer Anwendung erfolgt, sondern in einer Programmiersprache. Es können deshalb keine generalisierten Metaphern aus bestehenden, zweckbestimmten Systemen übernommen werden. Auch an die Benutzerschnittstelle werden besondere Anforderungen gestellt. Zum einen soll sich das System gezielt an Interface-Metaphern existierender Zeitleistensysteme anlehnen und somit in erster Linie ein gestalterisch, intuitives Arbeiten ermöglichen. Andererseits dürfen keine programmiertechnischen Spielräume verbaut werden, da das System auch eine völlig artfremde Nutzung erlauben muss. Zusätzlich muss, durch die Integration in ein bestehendes System, auch auf in VWV vorhandene Benutzungskriterien Rücksicht genommen werden.

Zur Unterstützung dieser Untersuchung wird ein Prototyp entwickelt, der die nachfolgenden vorgestellten Konzepte exemplarisch realisiert. Als prototypische Implementierung wird ein Knoten erstellt, der sich vollständig in die Entwicklungsumgebung VWV integriert. Der Knoten soll dabei um ein zusätzliches Fenster erweitert werden, in dem der grafische Editor platziert wird. Dies ist vergleichbar mit den, in Kapitel 2.2.5. vorgestellten, Renderern. Die Integration in den Datenstrom von VWV setzt voraus, dass auch technische Strukturen, z.B. bei der Konzeption der Datenhaltung, übernommen bzw. weitergeführt werden.

4.2. Allgemeine Anforderungen

Die primäre Anforderung ist die Entwicklung eines praxistauglichen Systems. Weitere, allgemeine Anforderungen sind:

Flexibilität
Eine der Grundvoraussetzungen für einen intuitiven Umgang mit dem System ist die Möglichkeit des spielerischen Ausprobierens. Es muss ohne lange Einarbeitung möglich sein, verschiedenste Anwendungen mit dem Knoten zu realisieren und nach dem Echtzeit-Ansatz von VVVV zu modifizieren. Auch die Ansteuerung von anderer Software, unabhängig von VVVV, soll in die Planung mit einbezogen werden.

Integration
Um eine vollständige Integration in VVVV zu gewährleisten, muss ein Knoten entwickelt werden. Der Knoten muss sich nahtlos in den Datenstrom von VVVV einfügen und sowohl technische, als auch gestalterische Strukturen aufnehmen und weiterführen.

Interface
Das System muss ein grafisches Benutzerinterface bieten, welches sowohl vorhandenen Gestaltungsmetaphern von VVVV, als auch von bekannten Zeitleistensystemen aufgreift. Es ist jedoch zu beachten, dass dieses nicht mit einer Standardsoftware, die für einen bestimmten Einsatzzweck entwickelt wird, gleichzusetzen ist. Dennoch müssen Grundkriterien wie klare Strukturierung, Übersichtlichkeit und leichte Verständlichkeit erfüllt werden.

Performance
Da die Zeitleistensteuerung Teil eines Echtzeitsystems wird, sind auch Merkmale der Ausführungsgeschwindigkeit zu berücksichtigen. So ist die schnelle Berechnung von diffizilen Algorithmen oftmals wichtiger als eine besonders schöne Gestaltung.

Skalierbarkeit
Ein modularer Aufbau der Programmierung, der sich an objektorientierten Kriterien orientiert soll es ermöglichen, weitere Funktionen einbinden zu können.

Systemunabhängigkeit
Bereits im Vorfeld wird ein möglicher Wechsel des Betriebssystems eingeplant. Es ist daher erforderlich, Technologien einzusetzen die Zukunftssicherheit garantieren und die Programmierung so zu gestalten, dass ein eventueller Wechsel keine Re-Implementierung des Knotens erforderlich macht.

4.3. Spezifische Anforderungen

Neben allgemeinen Anforderungen, die zum Teil schwer objektiv zu messen sind, soll das System aber auch einige konkrete Anforderungen erfüllen. Die nachfolgend aufgeführten muss das System implementieren.

Keyframes
Die zu entwickelnde Zeitleistensteuerung soll einen keyframebasierten Ansatz implementieren. Dazu muss es möglich sein, Keyframes zu erstellen und auf einem Zeitstrahl anzuordnen. Ferner soll Keyframes ein Dezimalewert zugewiesen werden können.

Spuren
Es sollen multiple Spuren erstellt werden können, die es erlauben gleichzeitig verschiedene Parameter zu steuern.

Interpolation
Es muss die Möglichkeit bestehen, Werte, die sich zwischen zwei Keyframes befinden, interpolieren zu können. Dazu sollen verschiedene Interpolationsarten, wie lineare Interpolation und die Interpolation auf Basis einer parametrischen Kurve, zur Auswahl stehen. Weiterhin soll es möglich sein, Keyframes, welche innerhalb einer Spur liegen, mit unterschiedlichen Methoden zu interpolieren.

Ablaufsteuerung
Die vorhandenen Spuren müssen mit Hilfe einer Ablaufsteuerung abspielbar sein. Dabei soll immer der Wert ausgegeben werden, der sich an der Position des Abspielzeigers auf der jeweiligen Spur befindet. Zusätzlich muss es möglich sein, diese Abspielsteuerung nicht nur über das grafische Interface des Knotens zu aktivieren, sondern auch direkt über den Datenstrom von VVV.

Automation
Es soll ein Konzept entworfen werden, welches es erlaubt, den linearen Ablauf einer Zeitleiste zu modifizieren. Es muss möglich sein, auf der Zeitachse einen Sprung zu einem gewünschten Zeitpunkt auszuführen. Dadurch soll es ermöglicht werden, sogenannte Loops[36] auf der Zeitleiste zu generieren.

Datenhaltung
Die Einstellungen, welche am System vorgenommen werden, müssen gespeichert und wieder geladen werden können.

Neben diesen Kriterien existieren natürlich noch viele weitere Anforderungen, die ein zeitleistenbasiertes System erfüllen könnte. Deshalb werden im Folgenden einige definiert, die aber nicht zwingend implementiert werden müssen.

Aufzeichnung

[36] Loops sind definierte Zeitbereiche die zyklisch abgelaufen werden.

Das automatisierte Erstellen von Keyframes, durch Daten die von VVVV an den Knoten gesendet werden, stellt sehr vielfältige Anwendungsbereiche in Aussicht und wäre damit eine nützliche Erweiterung.

Datentypen
Denkbar wären, neben Keyframes die Dezimalwerte erhalten, auch weitere zur Verwaltung und Ausgabe VVVV-spezifischer Datentypen.

Importieren
Das Importieren von Daten anderer zeitbasierter Anwendungen, z.B. eines Kalenders, könnte die Flexibilität des Systems erhöhen.

4.4. Abgrenzungskriterien

Da das System einen Ansatz verfolgt, der generelle Parameteränderungen erlaubt und damit kein reines Animationssystem entwickelt wird, gibt es Funktionalitäten, die zweckgebundene Zeitleistensysteme besser realisieren. So wird auf ein Konzept für Geschwindigkeitskurven verzichtet. Ferner sollen keine Interpolationsarten integriert werden, die explizit für Animationen im 3D-Raum gedacht sind.

4.5. Anwendungsbereiche

Der Hauptanwendungsbereich des Systems liegt in der Produktion von interaktiven Medieninstallationen. Hier sind zeitgesteuerte Manipulationen von rechnergestützter Hardware (z.B. Motoren), oder die Manipulation von grafischen Objekten denkbar. Da das System als Teil einer Programmiersprache verstanden wird, ist es weitestgehend der Kreativität des Benutzers überlassen, was er mit diesem System steuern möchte.

4.6. Zielgruppe

Das System richtet sich in erster Linie an VVVV-Programmierer. Sie lassen sich in drei Gruppen zusammenfassen. Die erste und auch größte Gruppe von Nutzern hat einen gestalterischen Hintergrund. Meist sind es Kommunikations– oder Interfacedesigner mit entsprechender Ausbildung. VVVV findet hier Anwendung für die Gestaltung von interaktiven Exponaten. Als zweite Gruppe lassen sich Medienkünstler einordnen. Sie nutzen VVVV oft für Video– oder Grafikkunst. Die dritte Gruppe bilden Nutzer mit eher technischem Hintergrund, beispielsweise Informatiker oder Veranstaltungstechniker. Hier kommt VVVV zum Einsatz, wenn es um ausgefallene Projektionen oder die Ansteuerung von Hardware geht. Diese Aufteilung in Benutzergruppen kann natürlich nicht pauschalisiert werden.

5. Systementwurf

Nachdem der derzeitige Stand in VVV betrachtet wurde und klar ist, welche Anforderungen an das zu implementierende System gestellt werden, ist es notwendig, einen geeigneten Systementwurf zu erarbeiten. Dieser garantiert eine reibungslose Implementierung. Grundlegend werden Entwurfskriterien festgelegt, die bei der Realisierung von Bedeutung sind.

Die Integration des Systems soll durch die Entwicklung eines Knotens erfolgen. Das garantiert eine optimale Eingliederung in den Datengraph und eröffnet zudem Möglichkeiten im Umgang mit Zeitleisten, die in anderen Systemen so nicht vorhanden sind. Als Verdeutlichung soll zunächst das Beispiel der Instanziierung dienen. Befindet sich die Zeitleistensteuerung innerhalb eines Knotens in VVV, so ist es ohne weiteres denkbar, multiple Instanzen von ihr anzulegen. Dadurch wird es möglich, mehrere Zeitleisten in Abhängigkeit voneinander zu setzen. In Kapitel 7 werden vorstellbare Szenarien anhand von praktischen Beispielen demonstriert.

Da es zur Implementierung praktisch immer mehrere Wege gibt, werden im Folgenden verschiedene Lösungsansätze betrachtet und auf ihre Eignung überprüft.

Auswahl der Integrationsmethode

Es wurden bereits verschiedene Wege aufgezeigt, mit denen eine Implementierung einer Zeitleistensteuerung möglich wäre. In Bezug auf die in Kapitel 4 definierten Anforderungen wird anschließend kurz begründet, welche der Methoden bei der Realisierung zum Einsatz kommt.

Die Methodik des Subpatching zeichnet sich eigentlich nur durch ihre Flexibilität aus. Mit dieser modularen Lösung wären interne Komponenten auf projektspezifische Aufgaben anpassbar. Die Implementierung durch Subpatching wird jedoch wegen der bereits diskutierten Faktoren nicht in Erwägung gezogen. Durch fehlende Interfacekomponenten und die sehr aufwendige Entwicklung ist eine benutzerfreundliche und vielseitig einsetzbare Anwendung nicht realisierbar. In Gesprächen mit den Entwicklern von VVV stellte sich heraus, dass Skalierbarkeit und Systemunabhängigkeit entscheidende Kriterien sind, die unbedingt gewährleistet werden müssen. Zudem wurde schon seit langem über

die Möglichkeit diskutiert, VVVV auch für andere Anwendungsfälle einfach und modular erweitern zu können. Das ist durch eine Implementierung in den Quellcode sicherlich nicht gegeben, da dieser weder öffentlich noch leicht verständlich ist. Deshalb fiel nach längerem Diskurs die Wahl der Implementierung auf die Nutzung einer Plug-In Schnittstelle. Diese soll es sowohl der Zeitleistensteuerung als auch späteren Anwendungen ermöglichen, einen klar definierten Zugriff auf die, zur Erstellung eines dedizierten Knoten notwendigen, Funktionalitäten von VVVV zu erhalten. Da eine solche Schnittstelle jedoch zu Beginn dieser Arbeit noch nicht existierte, wurde in Gesprächen mit den VVVV Entwicklern entschieden, sie parallel zur Zeitleistensteuerung zu realisieren. Diese Entscheidung bringt Konsequenzen in Bezug auf den Systementwurf und die spätere Implementierung mit sich. Es wurden klare Anforderungen, nicht nur an den Knoten, sondern auch an die Schnittstelle gestellt. Allerdings muss beachtet werden, dass sich spezifische Anforderungen an die Schnittstelle durchaus erst in der Phase des Systementwurfs ergeben können. Jene können sich natürlich wieder rückkoppelnd auf den Entwurf und die Implementierung des Knotens auswirken. Auch konnte die Schnittstelle zuvor keinem Praxistest unterzogen werden. Letztlich muss also auf beiden Seiten eine gewisse Flexibilität im Entwurf garantiert sein. Die Schnittstelle selbst ist nicht Teil dieser Arbeit.

5.1. Integration in den Datengraph

Um den Knoten entsprechend den Anforderungen aus Kapitel 4.2. zu integrieren, muss dieser mit anderen Knoten innerhalb des Datengraphs kommunizieren können. Das erfolgt, wie bei allen anderen Knoten, über Pins. Daher sind sowohl Input-, als auch Output-Pins erforderlich um Daten von VVVV zu empfangen, bzw. an VVVV senden zu können (Abb. 38).

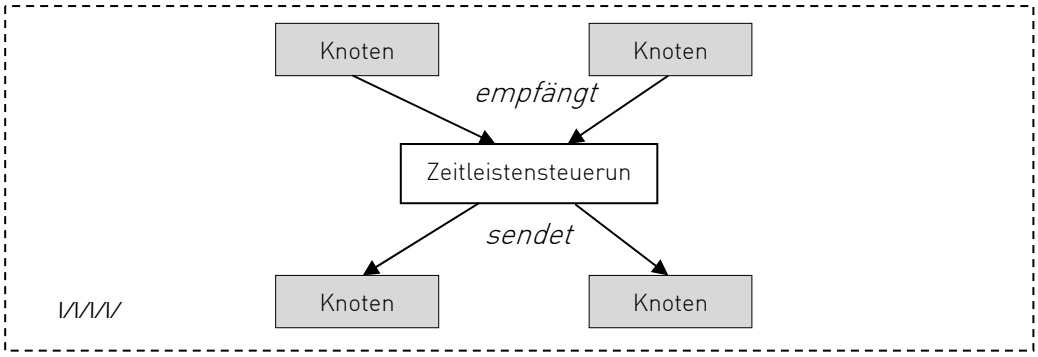


Abb. 38: Integration in den Datengraph von VVVV

Für spezifische Funktionen muss der Knoten unter Umständen auf seine eigenen Ausgabedaten zugreifen. Durch die Benutzung des in Kapitel 3.1.1. beschriebenen Knotens *FrameDelay* ist eine solche zyklische Verarbeitung möglich. (Abb. 39) Auf denkbare Anwendungsfälle wird im Verlauf der Arbeit eingegangen.

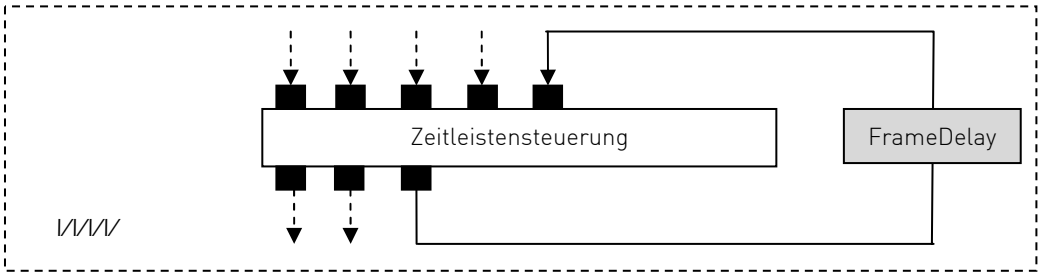


Abb. 39: Zyklische Verarbeitung von Ausgangsdaten

Die anzulegenden Input-Pins agieren alle in vordefinierten Wertebereichen. Es wird nicht nur der Datentyp eines Pins festgelegt, sondern auch der Minimal- und Maximalwert die der Pin verarbeiten kann. So wird, ähnlich der in Kapitel 2.2.3. beschriebenen Typsicherheit vermieden, dass das System in einen undefinierten Zustand gerät. Alle bei der Objektinstanziierung, anzulegenden Input-Pins werden, je nach Typ und Verwendungszweck, mit Standardwerten initialisiert. Diese Standardwerte sind als Konstanten festgelegt, werden aber überschrieben, sobald ein Benutzer sie manuell ändert. Da der zu entwickelnde Knoten nicht nur Daten von anderen Knoten empfängt und verarbeitet, sondern selbst eine Datenquelle ist, dürfen nachfolgende Knoten nicht in undefinierte Zustände geraten. Deshalb werden alle Output-Pins des Knotens ebenfalls mit Standardwerten initialisiert.

5.2. Systemarchitektur

Grundlegend besteht das zu entwickelnde System aus drei unterschiedlichen Ebenen (Abb. 40).

Die Ebene der Steuerung integriert die Komponenten der Abspiellogik. Außerdem wird hier die in Kapitel 4.3. definierte Automation realisiert. Sie arbeitet dabei autonom vom restlichen System. Die Verwaltungsebene ist als Kernkomponente der Anwendung spezifiziert. Sie organisiert die gesamte Kommunikation mit VVVV. So ist sie dafür verantwortlich In- und Output-Pins zu erstellen, die die Grundlage der Schnittstelle bilden. Auch die Datenhaltung ist ein Teil der Verwaltungsebene. Die Datenebene ist für die Handhabung von Datenspuren, Keyframes und deren Interpolationen zuständig. Im Folgenden werden grundlegende, funktionale Konzepte für die einzelnen Ebenen entwickelt. Zum besseren Verständnis wird mit der Datenebene begonnen.

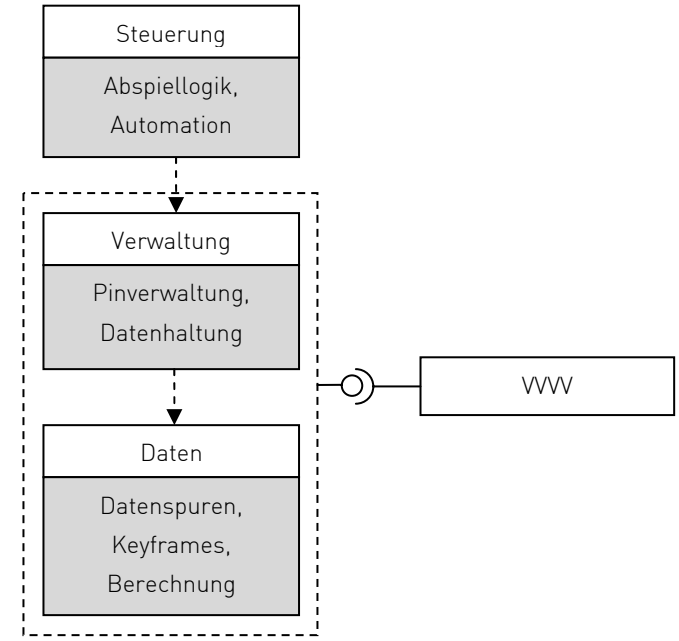


Abb. 40: Systemarchitektur

5.2.1. Datenebene

Innerhalb dieser Ebene sind alle Prozesse definiert, durch die direkt Daten erstellt, modifiziert und ausgegeben werden können.

5.2.1.1. Keyframes

Wie bereits erläutert, existieren in VVV fünf verschiedene Datentypen mit klar gesonderten Spezifikationen. Um die Integration verschiedener Datentypen zu gewährleisten ist es nötig, verschiedene Typen von Keyframes zu definieren. Jeweils für die Datentypen *value*, *string* und *color* sollen explizite Keyframe-Typen implementiert werden. Alle Typen besitzen die Eigenschaft Zeit, welche ihre Positionierung auf dem Zeitstrahl angibt. Dadurch erscheint es sinnvoll, sie von einer abstrakten Klasse abzuleiten. Der Parameter Zeit kann dabei bereits in der Basisklasse definiert sein. Durch dieses Konzept ist auch die spätere Implementierung weiterer Datentypen problemlos möglich.

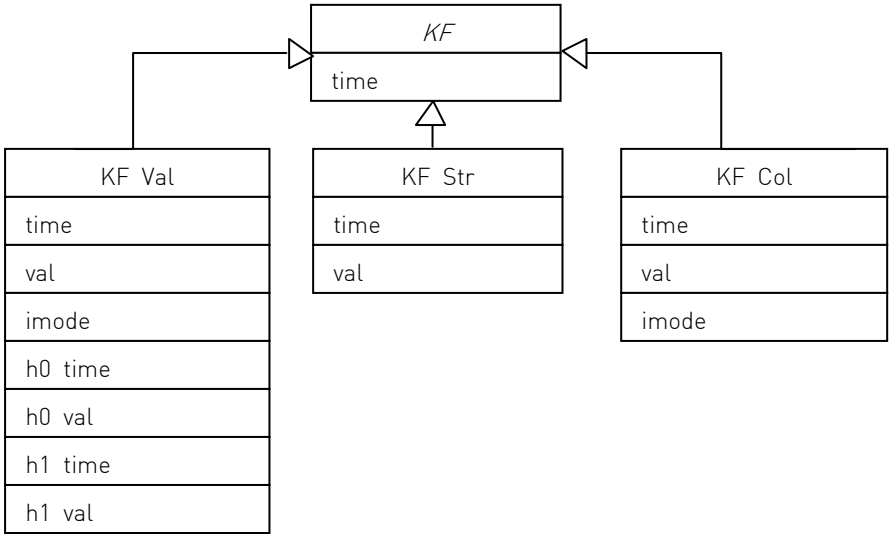


Abb. 41: Keyframe-Klassen

Anordnung

Da Werte in VVVV keinen Restriktionen unterliegen und der konkrete Anwendungszweck der Zeitsteuerung vorher nicht bekannt ist, soll auch die Anordnung der Keyframes auf dem Zeitstrahl keinen Begrenzungen unterliegen. In herkömmlichen Animationssystemen ist der Zeitstrahl, auf dem die Keyframes angeordnet werden, meist auf den positiven Bereich begrenzt. Diese Begrenzung entfällt hier jedoch. Das Intervall dem der Zeitstrahl und damit auch die Keyframes unterliegen ist theoretisch beschrieben durch:

$[-\infty, \infty] := \{t \in R \mid -\infty \leq t \leq \infty\}$

Praktisch unterliegt das Intervall der Grenze des Datentyps *double* und ergibt:

$[\text{min.}double, max.double] := \{t \in R \mid min.double \leq t \leq max.double\}$

Allerdings sollte das Intervall noch weiter eingeschränkt werden, da eine Erhöhung des Wertes *max.double* wiederum *min.double* zum Ergebnis hat.

Ein Keyframe kann somit im positiven als auch im negativen Zeitbereich angeordnet werden. Dies bietet die Möglichkeit, ohne die Einschränkungen eines Zeitstrahls mit halboffenem Intervall, editieren zu können.

Interpolationsmodi

In Kapitel 3.2. wurde bereits auf Verfahren der Interpolation zwischen Keyframes eingegangen. Dem Benutzer der Zeitleistensteuerung soll es möglich sein, zwischen verschiedenen Interpolationsarten zu wählen. Im Folgenden werden die zu implementierenden Verfahren diskutiert.

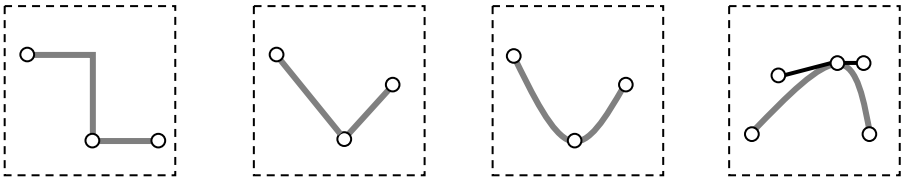


Abb. 42: Interpolationsmodi (von links) step, linear, spline, bézier

Step

Der Modus *step* (Abb. 42, links) stellt keine Interpolation im eigentlichen Sinne dar. Der Wert eines Keyframes wird solange ausgegeben, bis der Abspielzeiger das nächste Keyframe erreicht. Dieser Modus ist besonders gut geeignet um einfache Schaltvorgänge zu realisieren. In der Musikproduktion ist dieses Verfahren als Step Sequencing bekannt.

Linear

Dieser Interpolationsmodus (Abb. 42, mitte-links) integriert die bereits vorgestellte lineare Interpolation. Mit diesem Modus kann ein linearer Werteverlauf zwischen zwei Keyframes berechnet werden. Zu beachten ist, dass die Keyframes keinen Anstieg definieren. Dieser Modus ist sehr nützlich, wenn auch ein linearer Vorgang abgebildet werden soll. Dabei könnte es sich z.B. um die Lautstärkereglung eines physikalischen Gerätes handeln.

Spline und Bézier

Die Anforderungen in Kapitel 3.2.2. definieren die Interpolation anhand perimetrischer Kurven. Die Modi *Spline* und *Bézier* (Abb. 42, mitte-rechts und rechts) bieten einen stetigen Kurvenverlauf. Mit ihnen ist es möglich, weiche Übergänge an den Keyframes zu erzeugen. Das ist besonders für die Animation grafischer Objekte interessant. Da beide Modi auf derselben Berechnungsgrundlage implementiert werden, können sie im Folgenden zusammenhängend erläutert werden.

Das grundlegende Verfahren der beiden Modi ist die Bézier-Interpolation dritten Grades. Mit diesem Verfahren lassen sich durch das Anordnen der Tangentenvektoren alle erdenklichen Kurvenverläufe erstellen. Ist ein Keyframe im Modus *Bézier*, so werden dem Benutzer zusätzlich zwei Punkte visualisiert, mit denen er den Verlauf der Tangentenvektoren beeinflussen kann. Zur Berechnung der Kurvendarstellung kommt dabei der Algorithmus von *de Casteljau* zum Einsatz [vgl. Cubic07A]. Mit Hilfe einer rekursiven Ausführung linearer Interpolationen und Streckenunterteilung ist es möglich Punkte zu errechnen, die auf der Kurve liegen. So kann der gesamte Kurvenverlauf aus der X- und Y-Position der Stütz- und Tangentenpunkte errechnet werden [vgl. Watt92, S.74].

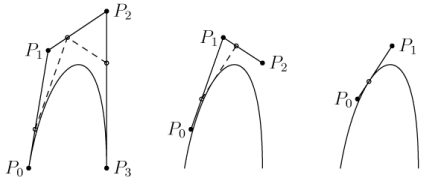


Abb. 43: Algorithmus von de Casteljau [vgl. Answ07A]

Das Spline-Verfahren ist dabei von der Bezierkurve abgeleitet, entsagt jedoch Tangentenvektoren. Diese werden wie bei Catmull-Rom Splines berechnet (siehe Kapitel 3.2.2.2.). Auf die Implementierung der TCB-Splines wird im ersten Schritt der prototypischen Implementierung verzichtet. Da kein reines Animationssystem entwickelt wird, besteht keine Notwendigkeit ein Verfahren zu implementieren, welches besonders auf die Drehung von grafischen Objekten im 3D-Raum zugeschnitten ist. Der Verzicht auf die lokale Kontrolle wird dadurch zunächst in Kauf genommen. Eine spätere Erweiterung durch TCB-Splines ist jedoch denkbar.

Nachdem die Interpolationsverfahren diskutiert wurden, gilt es, die verschiedenen Keyframe-Typen zu spezifizieren. Nicht jeder Typ implementiert auch jedes Interpolationsverfahren.

Value

Keyframes des Typs *value* können jeden beliebigen Dezimalwert im Bereich des Datentyps *double* aufnehmen. Sie implementieren alle der zuvor aufgezeigten Verfahren. Da jedes Keyframe-Objekt die Eigenschaft besitzt einen Interpolationsmodus zu speichern, kann die in Kapitel 4.3. definierte Anforderung, verschiedene Interpolationsverfahren miteinander zu kombinieren, erfüllt werden (Abb. 44). Das ermöglicht dem Benutzer eine sehr hohe Flexibilität bei der Erstellung eines Parameterverlaufs.

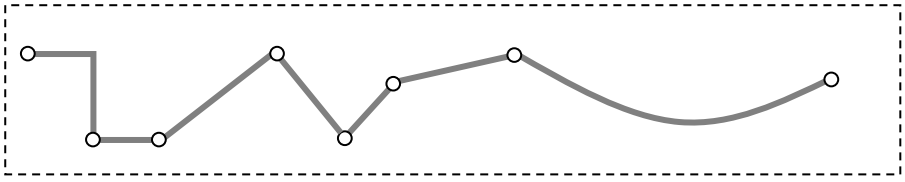


Abb. 44: Kombination von Interpolationsmethoden

String

Ein Keyframe des Typ *string* bildet auch den Datentyp *string* ab. So kann dieser Typ Zeichenketten speichern. Zur Ausgabe steht der Modus *step* zur Verfügung. Spätere Implementierungen anderer Interpolationsverfahren sind denkbar, erfordern jedoch eine explizite Untersuchung. Die weitere Ausführung dieser Problematik würde jedoch den Rahmen dieser Arbeit überschreiten.

Color

Der Keyframe-Typ *color* speichert einen dreidimensionalen Vektor. In diesem können Farbinformationen im Format RGB[37] gespeichert und mit den Modi *step* und *linear* interpoliert werden. Eine lineare Interpolation zwischen zwei Farben im RGB Farbraum würde jedoch immer direkt erfolgen und sich nicht an der physikalischen Farbwahrnehmung orientieren. Diese basiert auf der Wellenlänge des sichtbaren Lichts (Abb. 45). Um das Spektrum nachzubilden ist eine Interpolation im Farbraum HSL[38] besser geeignet. Abbildung 46 zeigt es an einer Beispielinterpolation zwischen den Farben Rot und Grün. Auf der linken Seite der Abbildung ist zu sehen, dass die Interpolation im RGB Farbraum nicht über die Farbe Gelb erfolgt, sondern direkt. Deshalb wird zunächst der RGB Wert in den Farbraum HSL konvertiert (Abb. 46, rechts). Das Ergebnis der durchgeführten Interpolation (die sich zunächst nur auf den Farbton beschränkt), wird anschließend wieder in den RGB Farbraum zurück konvertiert. Die Wahl einer geeigneten Interpolationsmethode für Farben ist immer vom konkreten Anwendungsfall abhängig. Deshalb muss es möglich sein, später weitere Modelle (z.B. RGB) hinzufügen zu können.

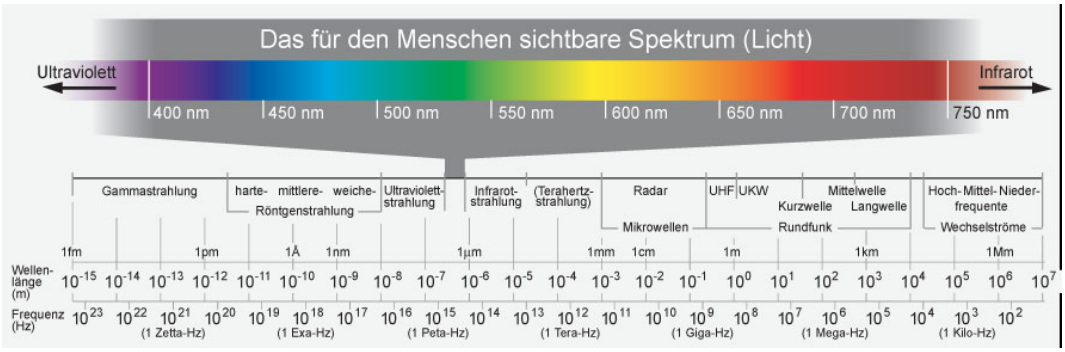


Abb. 45: Farbsektrum des sichtbaren Lichts, abhängig von der Wellenlänge [vgl. DR07]

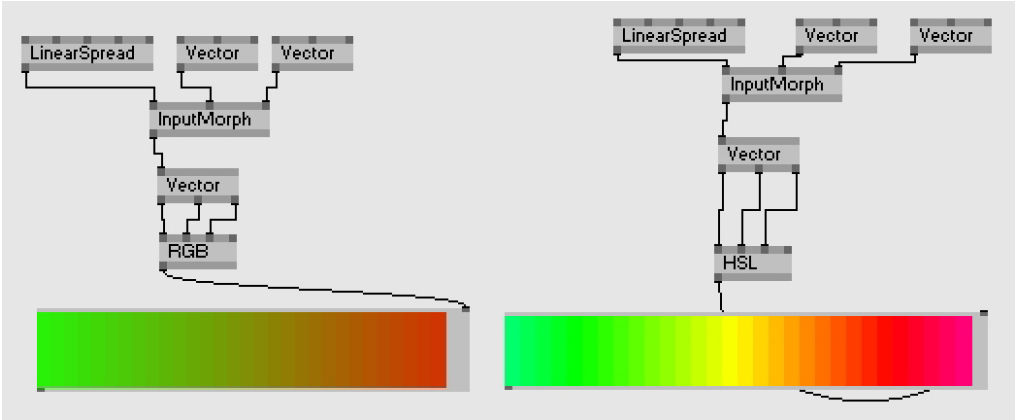


Abb. 46: Interpolationen, links im RGB Farbraum, rechts im HSL Farbraum

[37] Farbmodell, welches auf den Grundfarben (Rot, Grün, Blau) basiert.

[38] Farbmodell, welches auf Farbton (Hue), Sättigung (Saturation) und Intensität (Luminance) basiert.

5.2.1.2. Datenspuren

Laut der Anforderungen aus Kapitel 4 soll es möglich sein, mehrere Keyframe-Ebenen gleichzeitig zu bearbeiten. Dies wird durch die Datenspuren verwirklicht. In diesen werden die Keyframes sowohl grafisch erzeugt und bearbeitet, als auch die Interpolationsmethoden visualisiert (Abb. 47). Dabei sind Datenspuren typabhängig, d.h. es existiert für jeden Datentyp den das System verwalten kann auch genau ein Typ von Datenspur mit speziellen Attributen, sowie den entsprechenden Keyframe-Typen. Damit die Werte die durch die Keyframe-Interpolation erzeugt werden auch in VVVV weiterverwendet werden können, ist jede Datenspur auf spezifische Weise mit einem Output-Pin verknüpft. Diese Verknüpfung kann mittels verschiedener Methoden realisiert werden. Zum einen wäre es möglich, die Datenspuren als autonome Objekte zu betrachten, die ihre Werte typisiert an die zugewiesenen Output-Pins weiterreichen. Ebenso kann ein Output-Pin direkt als Datenspur abgebildet werden. Nun stellt sich die Frage, welche Implementierungsmethode im Hinblick auf die vorab definierten Anforderungen am besten geeignet ist.

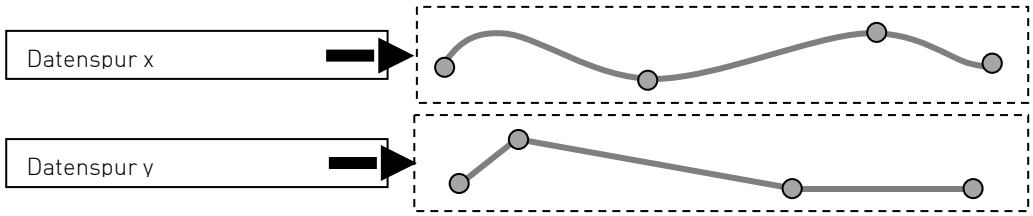


Abb. 47: Datenspuren enthalten Keyframe-Daten

Typisierte Zuweisung

Werden die Datenspuren als eigenständige Objekte angesehen, so existiert für jeden Typ Datenspur genau ein Output-Pin. Nach Initialisierung des Knotens existieren keine Datenspuren aber bereits statische Output-Pins für jeden Datentyp. Legt der Benutzer eine Datenspur an, so wird, je nach Typ der Spur, eine Verknüpfung zu dem entsprechenden Output-Pin hergestellt. Zum Schutz der Typsicherheit kann ein Pin immer nur Daten eines bestimmten Typs aufnehmen.

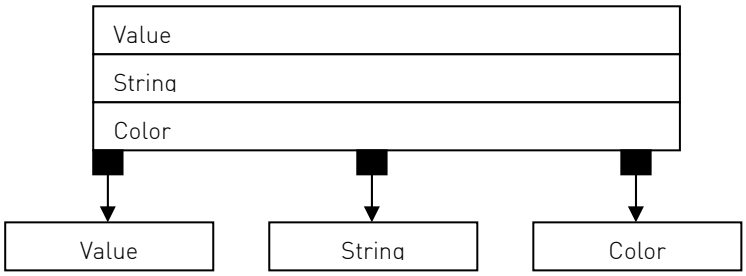


Abb. 48: Typisierte Zuweisung der Output-Pins

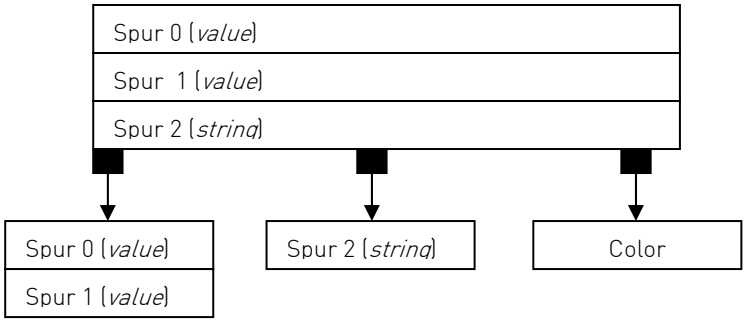


Abb. 49: Alle Datenspuren desselben Typs werden in einem Pin ausgegeben

Erfolgt das Anlegen einer zusätzlichen Datenspur, wird im entsprechenden Output-Pin ein neuer Slice erzeugt und mit der Spur verknüpft. Alle Spuren desselben Datentyps werden demnach in einem Output-Pin zusammengefasst (Abb. 49). Es existieren nur so viele Output-Pins wie unterschiedliche Datentypen. Der Vorteil dieser Methode ist die Einfachheit der Implementierung und Verwaltung, da lediglich einige statische Output-Pins angelegt werden müssten.

Explizite Zuweisung

Der alternative Ansatz sieht die Generierung jeweils eines Output-Pins pro Datenspur vor. Erstellt der Benutzer eine neue Spur, so wird ein kompatibler Output-Pin erzeugt. Im Gegensatz zur zuvor dargestellten Methode wird allerdings für jede weitere Spur ein neuer Output-Pin erstellt.

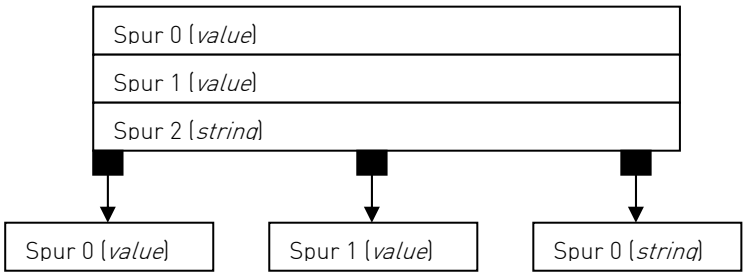


Abb. 50: Explizite Zuweisung der Output-Pins

Die Keyframes werden in diesem Modell nicht direkt von der Datenspur verwaltet. Zwischen ihr und der Keyframe-Ebene wird zusätzlich eine Zwischenschicht implementiert: die Slice-Ebene. Dem Benutzer ist es möglich, in jeder Spur manuell Slices anzulegen und somit eine Art „Sub-Spur" zu erzeugen. Diese Sub-Datenspuren existieren nicht nur virtuell im grafischen Modell, sondern bilden die realen Slices im zugewiesenen Output-Pin ab. Daher repräsentiert eine Datenspur immer einen vollständigen Output-Pin.

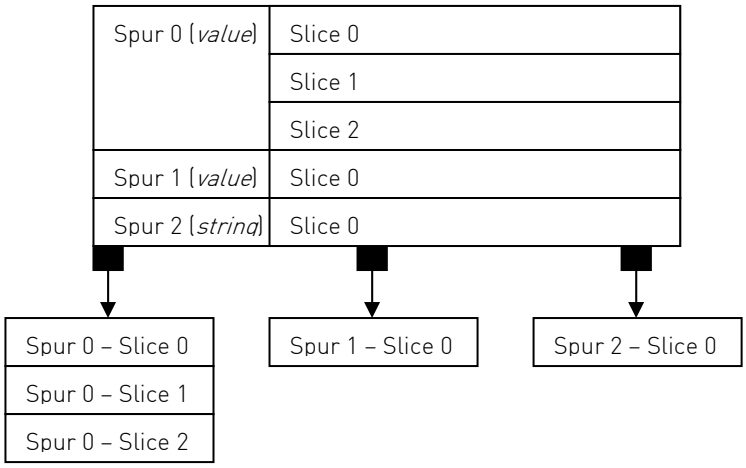


Abb. 51: Slices in der expliziten Zuordnung

Auswahl einer Methode

Die Methode des typisierten Austauschs zwischen Datenspuren und Output-Pins bietet den Vorteil, dass sie eine flache Hierarchie verwendet. Die Spuren verwalten direkt die Daten der Keyframes. Da für jeden Datentyp nur ein Output-Pin existiert, entsteht allerdings ein entscheidender Nachteil. Für jede weitere Datenspur eines schon vorhandenen Daten-Typs wird lediglich ein neuer Slice im entsprechenden Output-Pin erzeugt. Das macht eine spätere Zuordnung jedoch schwierig. Werden beispielsweise nur Werte bestimmter Datenspuren benötigt, müssen diese mühevoll aus dem Spread des jeweiligen Pins extrahiert werden. Es ist am Output-Pin nicht mehr abzulesen, welcher Slice welcher Datenspur zugeordnet ist. Der Benutzer müsste sich also die Zuordnung selbst merken. Das mag für eine überschaubare Anzahl von Spuren noch tragbar sein, wird aber mit steigender Anzahl zum Problem. Im folgenden Beispiel werden drei Datenspuren für eine Rotation (x,y,z) eines virtuellen 3D Körpers benötigt. Diese müssen unter Kenntnis der genauen Position mit Hilfe des Knotens *GetSlice* aus dem Output-Spread extrahiert werden. Ein weiterer Nachteil an dieser Methode zeigt sich, wenn eine der Spuren gelöscht wird. Die Zuordnungen stimmen nun nicht mehr überein. Es müssen alle Indizes des *GetSlice* Knoten manuell aktualisiert werden.

Die Methode der expliziten Zuweisung eines Output-Pins pro Datenspur bietet dagegen den Vorteil, dass sich der Benutzer zuvor überlegen kann, für welchen Zweck er eine Spur erzeugen möchte. Im Beispiel der Abbildung 52 würde er eine Spur explizit für eine Rotation erstellen. In dieser könnte er nun, je nach Bedarf, jeweils einen Slice für jede der Rotationsachsen erstellen. Das ermöglicht eine eindeutige Zuweisung des Output-Pins. Zusätzlich kann auf Grund der expliziten Zuordnung jeder Output-Pin mit einem eindeutigen Namen versehen werden (z.B. „Rotation"). Das erleichtert den späteren Arbeitsablauf, da der Name direkt am Pin des Knotens ablesbar ist. Soll nun zusätzlich eine Translation des Körpers ausgeführt werden, so kann bequem eine weitere Datenspur erzeugt und mit einem entsprechenden Namen versehen werden. Das bietet ein hohes Maß an Übersichtlichkeit. Wird eine Spur oder ein enthaltener Slice wieder gelöscht, wirkt sich dies lediglich auf den entsprechenden Output-Pin aus.

Der Nachteil dieser Methode ist der höhere Aufwand der Implementierung, da eine weitere Hierarchieebene eingefügt wird. Die Output-Pins müssen zudem dynamisch verwaltet werden. Um jedoch den Anforderungen nach Übernahme der in VVVV gängigen Strukturierung gerecht zu werden, wird die Methode der expliziten Zuweisung implementiert.

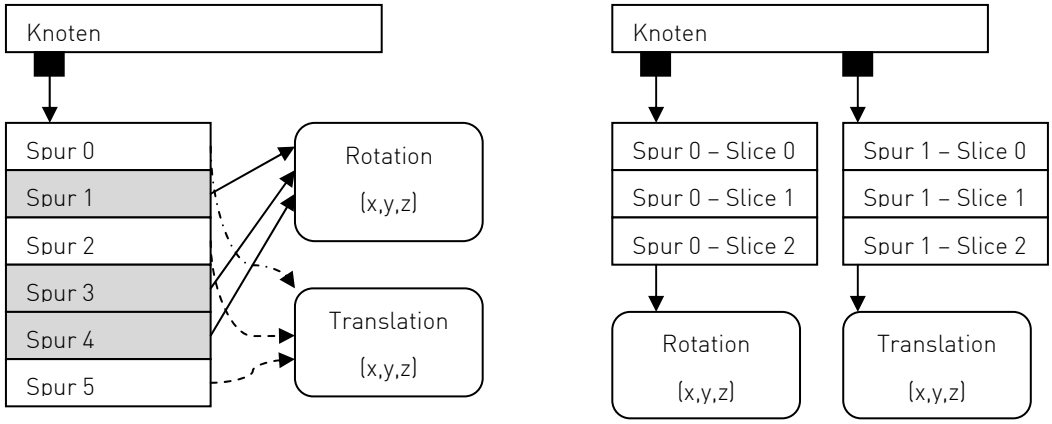


Abb. 52: Extrahieren von Werten, links: typisiert, rechts: explizit

Aufbau

Grundlegend betrachtet ist eine Datenspur eine virtuelle und visuelle Abbildung eines Output-Pins. Eine Datenspur ist dem entsprechend ein Container (Pin) in dem Sub-Spuren (Slices) existieren, die Keyframe-Objekte (Werte) enthalten. Dabei kann eine Spur beliebig viele Sub-Spuren umfassen, die wiederum beliebig viele Keyframe-Objekte enthalten können (Abb. 53).

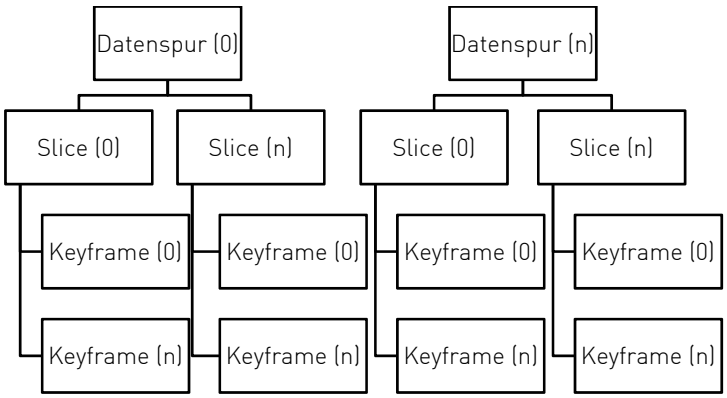


Abb. 53: Hierarchie der Datenspuren

Da alle Typen von Datenspuren die direkte Verknüpfung mit einem Output-Pin gemeinsam haben, werden sie auch von einer Basisklasse des Interfaces, die für das Erstellen eines Pins verantwortlich ist, abgeleitet (Abb. 54).

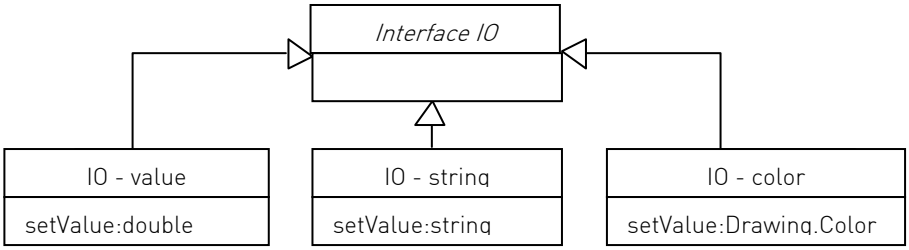


Abb. 54: Abgeleitete Pin-Typen auf denen die Datenspuren basieren

Durch diesen modularen Aufbau ist es möglich, später weitere Typen zu implementieren. Zusammenfassend kann der Prozess der Erzeugung von Keyframes und Datenspuren wie folgt beschrieben werden:

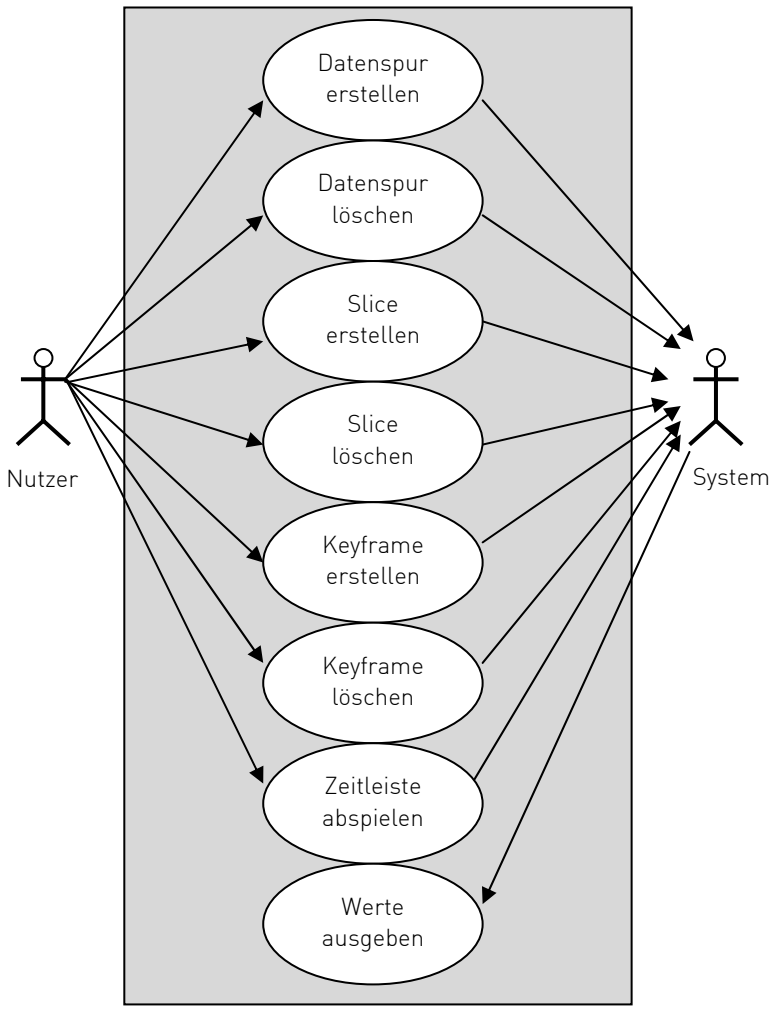


Abb. 55: Anwendungsfall Zeitsteuerung

5.2.1.3. Keyframe Generierung

Das System soll die Möglichkeit bieten, Keyframe-Werte von außen (in diesem Fall durch VVVV) einzugeben und aufzuzeichnen. Als Schnittstelle dient hierfür ein Input-Pin. Durch ihn kann innerhalb von VVVV eine Verbindung zu einem datenerzeugenden Knoten hergestellt werden. Um nun Keyframes in einer Datenspur generieren zu können, müssen eine oder mehrere Datenspuren visuell markiert werden. Für diese Funktionalität enthält jedes Slice einer Datenspur ein separates Interfaceelement. Zudem gibt es eine globale Aufnahmesteuerung die dasselbe Verhalten aufweist, wie die in Kapitel 5.2.2.1. beschriebene Abspielfunktionalität. Ist diese aktiviert, beginnt der Abspielzeiger (Abb. 57) die Zeitleiste abzulaufen. Zeitgleich wird ein Keyframe erstellt, das den aktuell am Input-Pin anliegenden Wert erhält. Damit nicht kontinuierlich neue Keyframes erzeugt werden, wird nur dann ein neues generiert, wenn sich der Wert am Input-Pin ändert (Abb. 56). Das reduziert das aufkommende Datenvolumen und verhindert zusätzlich redundante Daten.

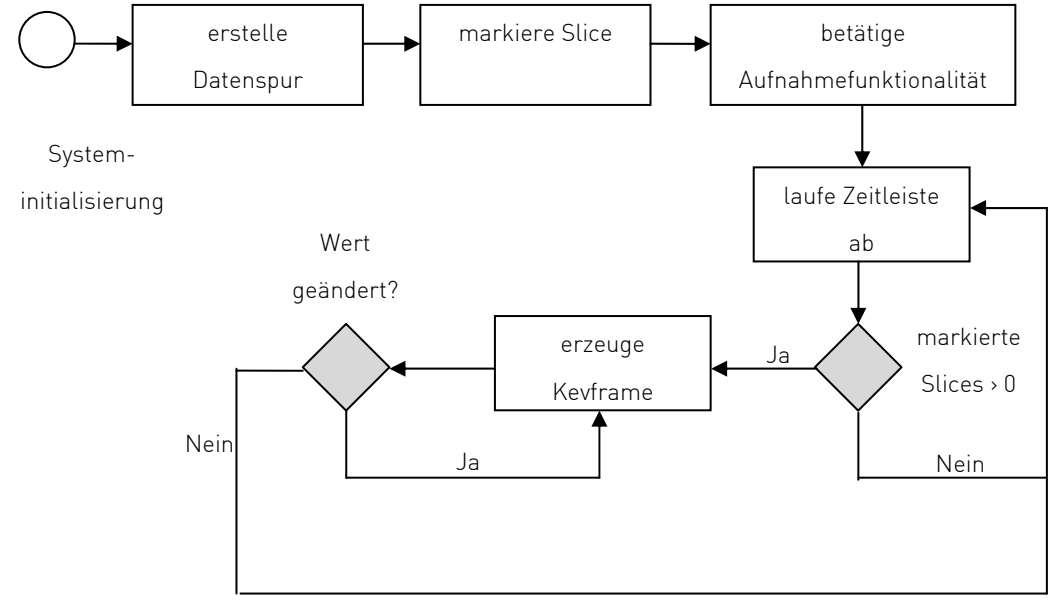


Abb. 56: Prozess der Keyframe Generierung

5.2.2. Steuerungsebene

Zu Beginn der Systemkonzeption stand die Überlegung das gesamte System in zwei getrennte Komponenten zu kapseln. Es wurde diskutiert, die Abspiellogik in einen separaten Knoten zu implementieren. Diese Kapselung sollte die Atomarität des Systems sicherstellen und damit das geforderte Aufgreifen von VVVV-interner Strukturierung gewährleisten. Es ist bei Knoten in VVVV üblich, nur eine klar definierte Funktionalität zu integrieren, damit nicht im Vorfeld der modulare Aufbau in der späteren Programmierung ausgeschlossen wird.

Das Verbinden der Interfaceelemente beider Knoten stellte sich hierbei jedoch als schwierig heraus. Der Knoten der Ablaufsteuerung hätte auch die Visualisierung des Zeitstrahls beinhalten müssen. Mit dieser Lösung wäre es jedoch unmöglich gewesen, in dem Knoten, der die eigentlichen Daten verwaltet, Keyframes auf dem Zeitstrahl anzuordnen, ohne dass die Knoten zumindest miteinander verbunden sind. Zudem wäre es für das grafische Interface ungeeignet zwei verschiedene Knoten zu nutzen um eine globale Systemanforderung zu erfüllen. Ablaufsteuerung und Datenebene werden deshalb in einem einzigen Knoten implementiert. Um jedoch nicht auf die Flexibilität einer modularen Lösung verzichten zu müssen, soll es das System ermöglichen, das Ablaufen der Zeitleiste sowohl durch den grafischen Editor des Knotens, als auch von außen (durch VVVV) steuern zu lassen. Das kombiniert die Freiheit zweier, voneinander getrennter Knoten mit der Möglichkeit alle Funktionalitäten in einem Knoten zu bündeln.

5.2.2.1. Funktionalitäten

Ablaufen der Zeitleiste

Eine Grundfunktionalität der Ablaufsteuerung ist das Ablaufen der Zeitleiste. Daher wird ein Abspielzeiger implementiert, der sich linear auf dem Zeitstrahl entlang bewegen kann. Die Geschwindigkeit mit der sich der Abspielzeiger bewegt, orientiert sich dabei an einer Abbildung der Systemzeit. Das heißt, im ersten Schritt der Implementierung wird sich der Abspielzeiger innerhalb einer Sekunde genau soweit auf dem Bildschirm bewegen, wie die visuelle Darstellung den Betrag des Vektors *s* definiert.

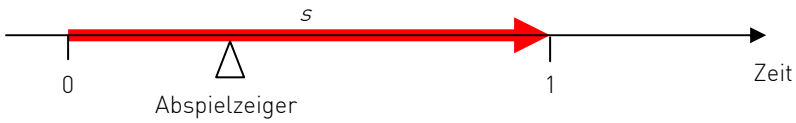


Abb. 57: Weg, den der Abspielzeiger zurücklegt

Wird die visuelle Darstellung nun vergrößert, vergrößert sich auch automatisch der Betrag des Vektors *s* und der Abspielzeiger legt einen längeren Weg zurück. Die Zeit, die er für den Weg benötigt, wird dabei jedoch nicht beeinflusst. Das ist mit der Maßstabsänderung auf einem Stadtplan zu vergleichen. Für spätere Erweiterungen wäre es denkbar, dass sich der Betrag des Vektors *s* auch getrennt von der visuellen Darstellung verändern ließe. So wäre es möglich, die Abspielgeschwindigkeit zu modifizieren.

Da VVVV das Basis-Framework des Systems bildet, wird auch die Systemzeit von VVVV zur Verfügung gestellt. Modifikationen in VVVV wirken sich direkt auf den Knoten aus.

Pausieren
Es soll jederzeit möglich sein den Abspielzeiger anzuhalten, um ihn später an derselben Stelle wieder ablaufen zu lassen.

Externe Kontrolle
Die Anforderung der Ablaufsteuerung definiert weiterhin eine Funktionalität, die es erlauben soll, die Ablaufsteuerung durch VWV zu kontrollieren. Um das zu ermöglichen, muss der Knoten einen Input-Pin zur Verfügung stellen der es erlaubt, einen Zeitwert durch VWV einzugeben. Dies könnte z.B. durch einen der in Kapitel 3.1.1. vorgestellten Knoten erfolgen.

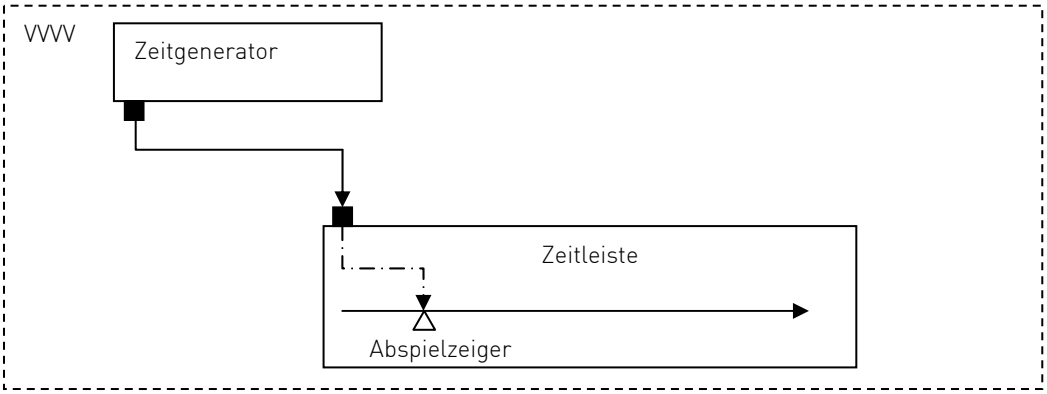


Abb. 58: Steuerung des Abspielzeigers durch VWV

Diese Möglichkeit der Steuerung eröffnet viele neue Wege im Umgang mit Zeitleisten, da der Abspielzeiger nicht mehr an eine Uhr gekoppelt ist, sondern sich z.B. auf Basis von mathematischen Berechnungen bewegen kann.

Die Funktionalität der externen Steuerung bietet allerdings noch eine weitere Art des Einsatzes, deren Möglichkeiten in Kombination mit einer so komplexen Programmierumgebung wie VWV nur erahnt werden können. Der Nutzer könnte zum Beispiel mehrere Zeitleisten hierarchisch anordnen. Es ist vorstellbar den Output-Pin einer Datenspur mit dem Input-Pin, der das Steuern des Abspielzeigers ermöglicht, einer anderen Zeitleisteninstanz zu verbinden. Daher wird es möglich, trotz der in Kapitel 4.6. definierten Abgrenzung keine Geschwindigkeitskurven zu implementieren, den zeitlichen Ablauf durch die Datenkurve einer anderen Zeitleisteninstanz zu beeinflussen. Theoretisch ist es auch denkbar durch das in Abschnitt 5.1. demonstrierte Verfahren die zeitliche Abfolge derselben Zeitleisteninstanz zu beeinflussen.

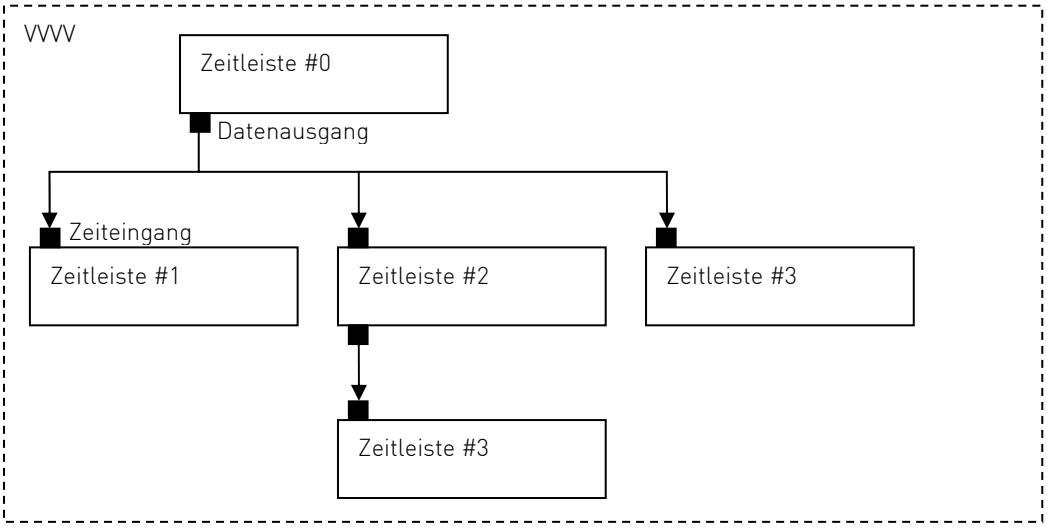


Abb. 59: Verknüpfung multipler Zeitleistensteuerungen

Automation

Nach den Anforderungen soll eine Funktionalität integriert werden, die es erlaubt, auf dem Zeitstrahl zu springen. Um diese Funktionalität zu ermöglichen, wird das Prinzip eines endlichen Automaten aufgegriffen. Die Automation ist hierarchisch über der Abspiellogik angesiedelt, da sie diese kontrollieren muss. Daraus ergibt sich der schematische Aufbau in Abbildung 60.

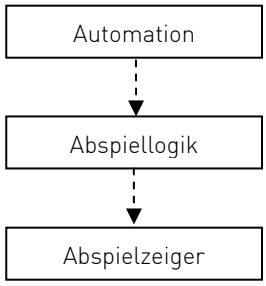


Abb. 60: Ablaufsteuerung

Zustände, Zustandsübergänge, Ereignisse

Ein endlicher Automat besteht aus einer Menge von Zuständen, Zustandsübergängen und Ereignissen. Diese Elemente realisieren in Kombination mit dem Zeitstrahl die gefordert Funktionalität. Das System ermöglicht dem Benutzer das Platzieren von Zuständen auf dem Zeitstrahl. Dafür wird im Benutzerinterface ein spezieller Bereich definiert. Es können beliebig viele Zustände erzeugt und auf dem Zeitstrahl angeordnet werden. Diese benennt das System automatisch und eindeutig. Der zeitliche Bereich zwischen zwei aufeinanderfolgenden Zuständen symbolisiert den Zustandsübergang (Abb. 61).

Abb. 61: Zustände und Zustandsübergang auf dem Zeitstrahl.

Befindet sich der Abspielzeiger zwischen einem Zustand *n* und einem Zustand *n+1* so ist ein Zustandsübergang erreicht. Um nun die Sprungfunktionalität zu realisieren, werden sowohl Ereignisse, als auch Aktionen benötigt. Da Ereignisse immer mit Zuständen verbunden sind, müssen diese explizit für einen Zustand erstellt werden. So ist es möglich, jedem Zustand ein Ereignis zuzuweisen.

Abb. 62: Zustände mit dazugehörigen Aktionen

Dazu steht ein kleines textuelles Eingabefeld bereit, das verfügbar wird sobald ein Zustand mit der Maus selektiert wurde. Dieses Eingabefeld akzeptiert einen Ausdruck mit dem angegeben werden kann, auf welches Ereignis der gewählte Zustand reagieren soll und welche Aktion daraufhin ausgelöst wird. Da der Benutzer mit der Maus lediglich Zustände erzeugen kann galt es zu überlegen, wie mit einer möglichst einfachen Syntax eine Verknüpfung von Zustand, Ereignis und Aktion ausgedrückt werden kann.

Der Ausdruck, mit dem die Zuweisung erfolgt; hat die folgende Form:

Ereignis = Sprungziel (event name = state name)

Dadurch wird ein Ereignis mit einer Aktion verknüpft und dem ausgewählten Zustand zugewiesen. Die Aktionen die das System ausführen kann, beschränken sich im ersten Schritt der Implementierung auf einen Sprung zu einem anderen Zustand. Der Name des Zustands; zu dem gesprungen werden soll; wird auf der rechten Seite des Zuweisungszeichens angegeben. Da die Namen der Zustände eindeutig sind, existiert für jeden Zustandsnamen genau ein Zeitpunkt auf dem sich dieser Zustand gerade befindet. Zum Ausführen des Sprungs bedarf es jedoch noch eines Ereignisses. Dessen Name kann frei vom Benutzer gewählt werden und befindet sich auf der linken Seite des Zuweisungszeichens. Sobald der Ausdruck mit der Return-Taste bestätigt wird; erfolgt zunächst eine Überprüfung mit Hilfe eines regulären Ausdrucks. Ist der Ausdruck valide, wird am Knoten ein Input-Pin erzeugt, der den Namen des Ereignisses trägt. Dieser Pin kann nur den Wert *0* oder *1* annehmen. Damit wird es möglich, durch VVVV ein Ereignis auszulösen.

Ist die Ablaufsteuerung aktiv, wird geprüft ob sich der Abspielzeiger auf der Position eines Zustandes befindet. Trifft das zu und ist der Ereignis-Pin dieses Zustandes *1*, springt der Abspielzeiger zu der Position des Zustandes, der als Sprungziel definiert wurde.

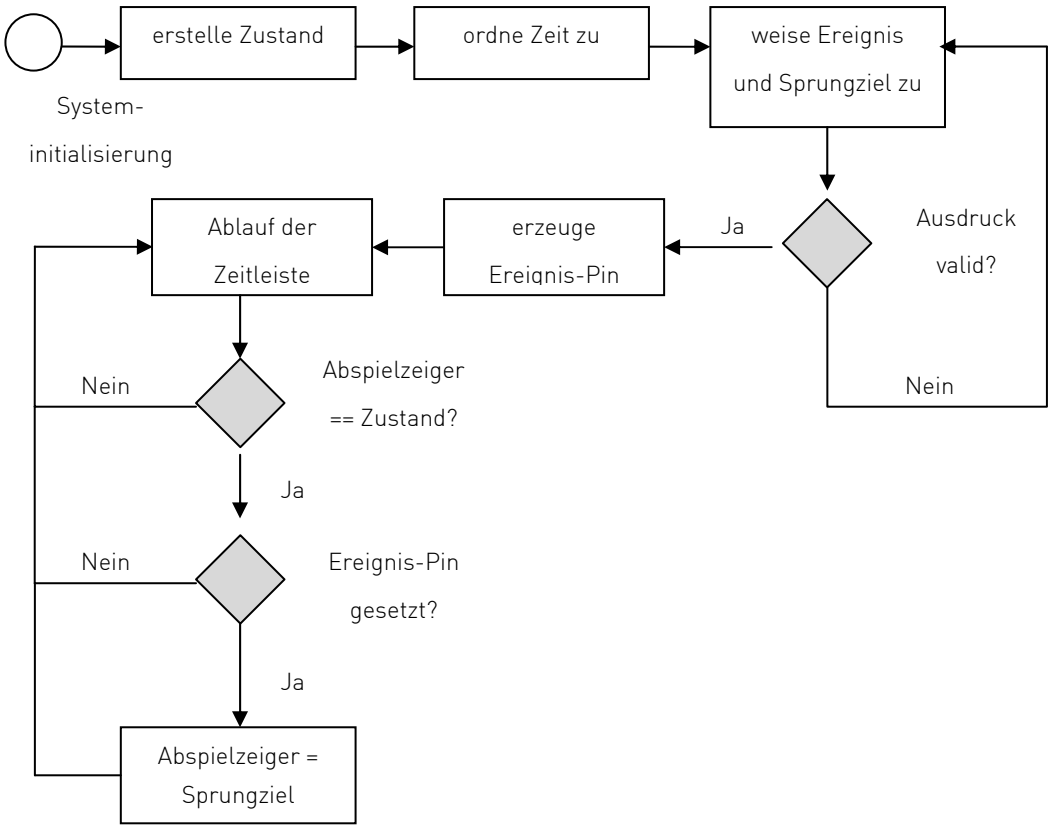


Abb. 63: Ablauf der Automation

Am Knoten der Zeitsteuerung existiert zusätzlich ein Output-Pin (*Seek*), der immer dann von *0* auf *1* schaltet, wenn der Abspielzeiger zu einem Zustand springt. Damit ist es möglich, ähnlich dem Konzept der Bangs in Max (Kapitel 3.3.2.), abhängigen Knoten mitzuteilen, ob auf der Zeitleiste gesprungen wird.

Da jedoch nur geprüft wird, ob sich die Position des Abspielzeigers direkt über einem Zustand befindet, entfernt sich dieses Konzept im ersten Implementierungsschritt etwas von einem endlichen Automaten. Dieser würde die Aktion auslösen, sobald sich der Abspielknopf in einem Übergang zwischen zwei Zuständen befindet. Für die Zukunft ist geplant der Automation auch dieses Verhalten zuordnen zu können.

Berechnung

Für die Berechnung eines Wertes anhand der Position des Abspielzeigers wird jeweils das Keyframe-Paar $KF_n(time) < cursor(time) < KF_{n+1}(time)$ betrachtet. Der Ausdruck beschreibt das Keyframe, das sich zeitlich vor und hinter dem Abspielzeiger befindet. Zunächst wird der Abstand zwischen den Keyframes errechnet und dann in das Intervall [0,1] abgebildet.

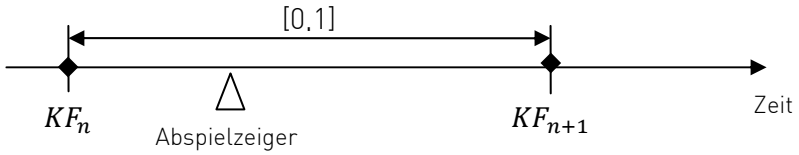


Abb. 64: Keyframes vor- und nach dem Abspielzeiger

Der Abspielzeiger bewegt sich dadurch in dem Intervall [0,1] und ergibt den Parameter *t*, der zur Berechnung der eigentlichen Interpolation dient. Im Fall einer linearen Interpolation ergibt sich daraus:

$$Q(t) = (1-t)KF_n(value) + t(KF_{n+1}(value))$$

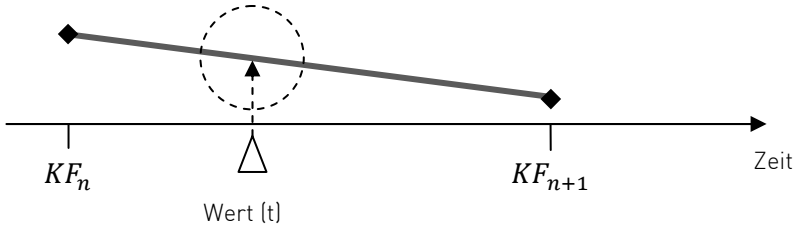


Abb. 65: Berechnung des Wertes zwischen zwei Keyframes

Bei der kubischen Interpolation beschreibt *t* den prozentualen Abstand zwischen dem ersten und letzten Kontrollpunkt der Kurve. Eine allgemeine Formel für die Berechnung einer parametrischen Funktion lässt sich wie folgt definieren:

$$B_{i,n}(t) = \frac{n!}{i!\,(n-i)!}(1-n)^{n-i}t^i$$

In der Formel definiert *n* den Grad der Kurve.

5.2.3. Verwaltungsebene

Die wichtigste Funktionalität der Verwaltungsebene ist die Schnittstelle zwischen VVV und der Zeitleistensteuerung. Hier werden die In- und Output-Pins erstellt und verwaltet, die der Knoten benötigt um Daten von VVV entgegen zu nehmen oder an VVV zu senden. Zudem zeichnet die Verwaltungsebene für die Datenspuren und die Datenhaltung verantwortlich.

Pins

Der zu implementierende Knoten unterscheidet verschiedene Arten von Pins, die alle von einem datentypabhängigen Basis-Pin abgeleitet werden. Dieser Basis-Pin wird von der Schnittstelle zur Verfügung gestellt. Von ihm können In- und Output-Pins abgeleitet werden und mit diversen Eigenschaften, z.B. Name oder Dimension, auf einen bestimmten Zweck angepasst werden. Neben In- und Output-Pins unterscheidet das System zwischen statisch- oder dynamisch erzeugten Pins. Als statische Pins werden jene bezeichnet, die bereits mit der Objektinitialisierung erzeugt werden und auch nicht vom Benutzer gelöscht

werden können. Als dynamisch werden in dieser Arbeit Pins verstanden, die zur Laufzeit vom System angelegt oder gelöscht werden. Dies basiert zumeist auf einer Interaktion mit dem System.

Datenhaltung

Es wurde spezifiziert, dass es möglich sein soll, relevante Daten auch speichern und laden zu können. Zu diesem Zweck wird eine Technik eingesetzt, die bereits in VVVV Verwendung findet: das Speichern von Daten in Input-Pins. Da ein Pin nicht nur einzelne Daten sondern auch Spreads aufnehmen kann ist es möglich, Listen von Werten in einen Pin zu schreiben. Dies ist mit einfachen Tabellen einer Datenbank vergleichbar. Lediglich die Logik zur Datenverwaltung fehlt. Wird ein Patch innerhalb von VVVV gespeichert, so werden auch alle an den Input-Pins anliegenden oder manuell eingestellten Werte gespeichert. Diese sind wieder verfügbar wenn der Patch geladen wird. Wird durch den Knoten zur Laufzeit ein Input-Pin erzeugt und mit Werten beschrieben, so ist dieser Pin nach dem Speichern und Laden des Patches wieder abrufbar und kann ausgelesen werden. Um diese Technik für den zu entwickelnden Knoten zu verwenden, muss sichergestellt werden, dass alle zu speichernden Daten jederzeit in Input-Pins bereit liegen. VVVV sendet kein Event beim Speichern des Patches an die Knoten. Daher ist es nicht möglich innerhalb des Knotens festzustellen, wann der Patch gespeichert wird. Jede Aktualisierung muss also direkt in den zuständigen Input-Pins erfolgen.

Wird ein Input-Pin speziell zum Speichern von Werten erzeugt, muss verhindert werden, dass der Benutzer diesen Input-Pin manuell verändern, bzw. durch Anlegen einer Verbindung zu einem anderen Knoten überschreiben kann. Um das zu realisieren verfügt ein Pin über spezielle Parameter.

Configuration Pin

Ein Pin der nicht am Knoten selbst sichtbar ist, sondern lediglich im Inspektormenü (siehe Kapitel 2.2.3.). Durch diesen können zwar keine Verbindungen zu anderen Knoten bestehen, aber der Benutzer kann noch immer manuell auf den Wert eines Slice Einfluss nehmen. Dieser Pin-Typ eignet sich daher nur bedingt für das Speichern von Daten und wird eher verwendet um die Komplexität eines Knotens zu verbergen.

Hidden Pin

Mit Hilfe dieses Parameters ist es möglich einen Pin zu verstecken. Er taucht weder am Knoten selbst, noch im Inspektormenü auf. Diese Methode eignet sich sehr gut um systeminterne Daten zu speichern.

Ein großer Vorteil dieser Methode ist, neben der VVVV Konformität die Möglichkeit, in späteren Versionen bestimmte Pins der Datenhaltung für den Benutzer sichtbar zu machen. So ist es z.B. vorstellbar Keyframe-Positionen mit einem VVVV Patch zu erstellen oder zu manipulieren.

5.3. Benutzerschnittstelle

Die Benutzerschnittstelle besteht aus zwei Komponenten: Zum einen aus den Interaktionskonzepten, zum anderen aus der grafischen Repräsentation.

5.3.1. Grafische Repräsentation

Grafisch ist die Zeitleistensteuerung in zwei Teile gegliedert. Einerseits existiert die Darstellung des Knotens im Datengraph von VVVV. Andererseits wird ein separates Fenster erzeugt, in dem der grafische Editor abgebildet wird. Es wurde Wert auf eine klare Gestaltung gelegt, da dies die einfachste und stärkste Methode ist, komplexe Sachverhalte zu kommunizieren. [vgl. Tufte01, S.9]

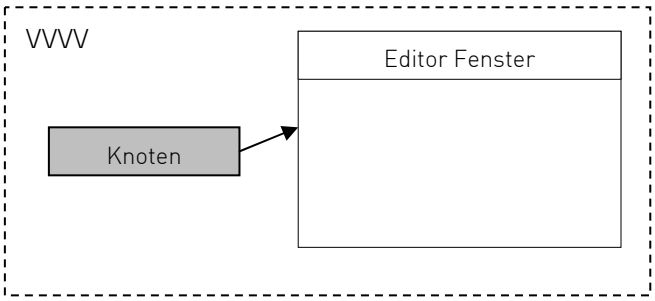


Abb. 66: Grafische Anordnung in VVVV

Wie Abbildung 67 zeigt, ist die grafische Benutzerschnittstelle in vier Grundbereiche aufgeteilt.

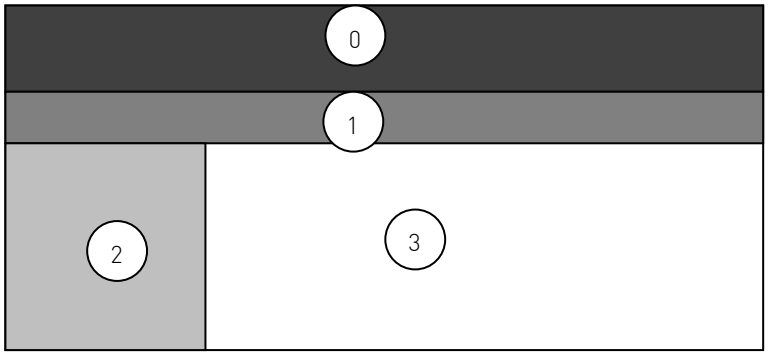


Abb. 67: Grundstruktur der grafischen Benutzerschnittstelle

Die Bedienelemente der Abspielsteuerung sollen sich im oberen Abschnitt (0) befinden, da sie die globale Kontrolle der Zeitsteuerung repräsentieren. Weiterhin finden sich hier die Funktionalitäten, die zum Erstellen der Datenspuren notwendig sind. Im mittleren Bereich (1) wird der Zeitstrahl mit Hilfe einer Zeiteinteilung abgebildet. Außerdem sind die Elemente der Automation auf dem Zeitstrahl definiert. Das untere Feld ist in zwei Teilabschnitte gegliedert. Auf der linken Seite (2) befinden sich die Bedienelemente mit denen sich die Eigenschaften der Datenspuren modifizieren lassen. Im rechten Teil (3) werden

die Datenspuren sowie die in ihnen enthaltenen Slices und Keyframes visualisiert. Eine Darstellung der grafischen Benutzerschnittstelle zeigt Abbildung 68.

5.3.2. Interaktionskonzepte

Aus den Anforderungen geht hervor, dass auch im Bezug auf die Interaktion bisherige Strukturen aus VWV übernommen werden sollen, um eine optimal Integration in die Entwicklungsumgebung zu garantieren. Die Übernahme bezieht sich auf allgemeine Bedien- wie auch grafische Konzepte.

Funktionalitäten
Funktional werden die Komponenten aus Abbildung 68 wie folgt belegt:

- *0*
  Bedienelemente der Abspielsteuerung: Abspielen; Pausieren; Aufnehmen; Zurücksetzen
- *1*
  Anlegen von Datenspuren der Typen: *value*; *string*; *color*
- *2*
  Zeitstrahl; Kann nach links und rechts verschoben werden. Weiterhin ist eine Zoomfunktionalität integriert, mit der ein Zeitabschnitt vergrößert oder verkleinert werden kann. Anlegen-, Verschieben- und Löschen von Zuständen
- *3*
  Eingabe eines Ausdrucks für einen selektierten Zustand; Ausgabe der aktuellen Position des Abspielzeigers
- *4*
  Benennen der Datenspuren; Ein- und Ausklappen der Spuransicht.
- *5*
  Datenspur-spezifische Funktionalitäten; Slices anlegen, zur Aufnahme markieren, Wertebereiche festlegen
- *6*
  Erzeugen-, Verschieben- und Löschen von Keyframes; Multiple Selektion; Festlegen des Interpolationsmodus.
- *7*
  Positionierung des Abspielzeigers

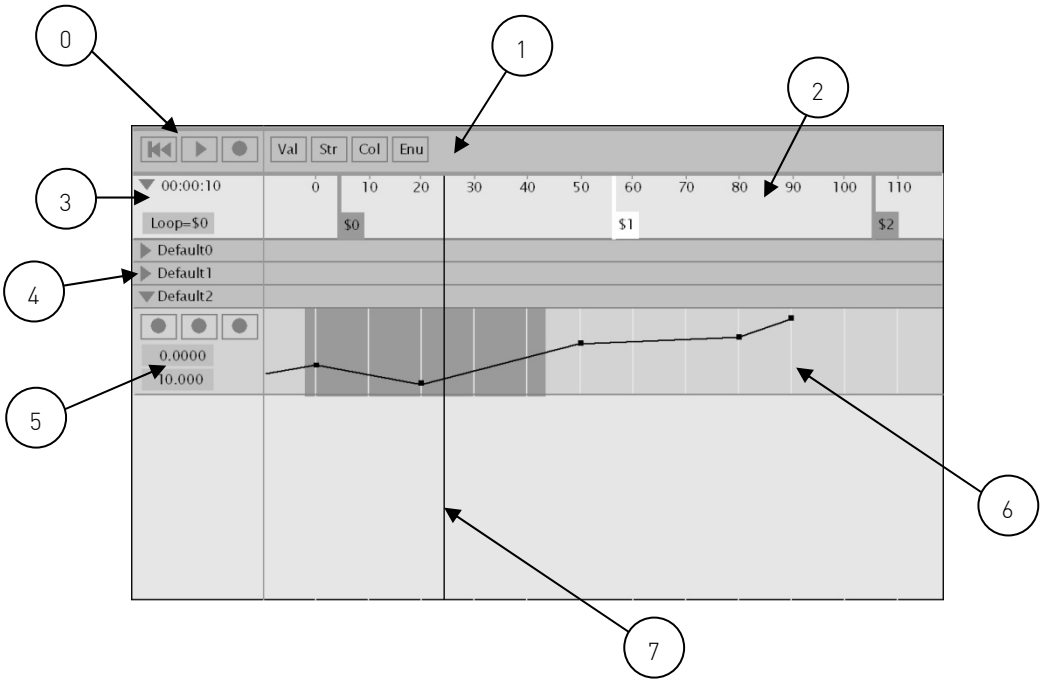


Abb. 68: Komponenten der grafischen Benutzerschnittstelle

Maussteuerung

Die meisten Interaktionen mit der Zeitleistensteuerung basieren auf einer Interaktion mit Hilfe der Maus. In VVV gelten verschiedene Konventionen im Umgang mit der Maus, die auch für die Zeitleistensteuerung übernommen werden sollen. Die rechte Maustaste bildet immer eine Manipulation in Kombination mit der X- oder Y-Verschiebung ab. So kann der Inhalt eines Patchfensters (Kapitel 2.2.2.) durch einen Rechtsklick, in Kombination mit einer Bewegung, verschoben werden. Auch die IO-Box erlaubt die Manipulation von Werten durch diese Interaktion. Ein Klick auf die mittlere Maustaste öffnet in VVV das Hauptmenü. Diese Funktion sollte in der Zeitleistensteuerung nicht zum Einsatz kommen.

Tastatursteuerung

Einige Funktionalitäten der Zeitleistensteuerung werden mit Hilfe von Tastaturkommandos zur Verfügung gestellt. Das Abspielen der Zeitleiste kann durch einen Tastendruck erfolgen. Auch das Löschen von Keyframes geschieht durch eine Tastatureingabe. Ferner kann ein selektiertes Keyframe mit Hilfe der Pfeiltasten in kleinen Schritten bewegt werden. Es ist darauf zu achten, keine Tastaturkommandos einzusetzen, die bereits in VVV definiert sind.

IO-Box

Das bereits vorgestellte Konzept der IO-Box wird für die Zeitleistensteuerung übernommen. Da nicht alle Funktionalitäten benötigt werden, wird eine Variante implementiert, die auf das Manipulieren von Dezimalwerten und Zeichenketten beschränkt ist.

6. Realisierung

Im Folgenden wird die Implementierung des zuvor entworfenen Konzeptes partiell beschrieben. Einführend wird ein Überblick über die verwendeten Technologien gegeben. Besonderes Augenmerk gilt hierbei der Schnittstelle zu VVVV.

6.1. Sprachwahl und Technologien

Da zu Beginn dieser Arbeit keine Plug-In Schnittstelle zu VVVV existierte, galt es, im Diskurs mit den Entwicklern von VVVV eine Auswahl für die Programmiersprache zu treffen. Einer der wichtigsten Faktoren bei der Wahl der Programmiersprache war die gewünschte Skalierbarkeit und Systemunabhängigkeit. (Kapitel 4) Aber auch die Performance spielte eine entscheidende Rolle. Dadurch verengte sich die Auswahl schnell auf zwei Varianten: das objektorientierte C++ und die .Net[39] Sprache C# (ausgesprochen „C-Sharp"). Da die Plug-In Schnittstelle auch für weitere Projekte und Entwicklungen genutzt werden soll, ist die Zukunftssicherheit der Technologie eine weitere Voraussetzung. Schlussendlich spielten aber auch der Komfort und die Einfachheit des Programmierens eine wesentliche Rolle für die Entscheidung, den zu entwickelnden Prototyp sowie die Plug-In Schnittstelle in C# zu implementieren. Zudem wird die Typsicherheit der Sprache als großer Vorteil angesehen [vgl. Drayton03, S.5].

Als Entwicklungsumgebung kam dabei die freie IDE[40] SharpDevelop[41] in der Version 2.0 zum Einsatz. Zusätzlich ist es für C# notwendig, die Laufzeitumgebung .Net in der Version 2.0 zu installieren. Diese ist frei verfügbar. Durch die Nutzung der .Net Technologie ist die definierte Anforderung nach Systemunabhängigkeit nur teilweise erfüllt. Da die .Net Laufzeitumgebung,

[39] Technologie, die es ermöglicht, Programme in einer virtuellen Maschine ausführen zu lassen.
[40] Entwicklungsumgebung (integrated development environment)
[41] Weitere Informationen unter: http://www.icsharpcode.net

ähnlich wie Java, eine virtuelle Maschine abbildet, in der die Anwendung ausgeführt wird, muss beim Programmieren der Anwendung nicht auf betriebssystemspezifische Eigenheiten geachtet werden. Theoretisch wäre der Wechsel des Betriebssystems mit einer, für das gewünschte System kompilierten Laufzeitumgebung von .Net problemlos möglich. Leider bietet der .Net-Entwickler Microsoft nur eine Laufzeitumgebung für sein eigenes Betriebssystem Windows an. Um trotz dieses großen Mankos nicht auf die Vorzüge von C# verzichten zu müssen, kommt die freie Entwicklungs– und Laufzeitumgebung Mono[42] zum Einsatz. Mono bildet nahezu alle Funktionen der .Net Laufzeitumgebung der Version 2.0 ab und ist dabei für die Betriebssysteme Microsoft Windows, Linux, Unix und Mac-OS verfügbar. Mono ist ein Open-Source Projekt und wird damit auch dem Wunsch nach Zukunftssicherheit gerecht. Zur Versionskontrolle kam Tortoise SVN[43] zum Einsatz.

6.1.1. Beschreibung der Plug-In Schnittstelle

Die Plug-In Schnittstelle besteht grundlegend aus zwei Komponenten. Eine Komponente bildet der sogenannte Host, gemeint ist die Applikation, die die Zeitleistensteuerung ausführt, in diesem Fall VVVV. Es ist allerdings auch möglich, die Hostapplikation austauschbar zu gestalten. Das System wird so der Anforderung gerecht, auch mit anderen Anwendungen einsetzbar zu sein. Zum Testen der Applikation wurde nicht VVVV selbst genutzt, sondern eine Applikation die einen VVVV Knoten simuliert. (Abb. 69) Dieser Stand-Alone Host bildet dabei alle Funktionen eines Knotens in VVVV ab. Input- und Output-Pins können erzeugt und mit Daten beschrieben werden. Diese Variante wurde zum einen deshalb gewählt, weil die Implementierung der Schnittstelle in VVVV zum Zeitpunkt der Entwicklung noch nicht abgeschlossen war. Zum anderen ließ sich diese Host-Applikation direkt aus dem SharpDevelop Projekt kompilieren und ausführen. Das ersparte das Kopieren des erstellten Plug-Ins und das ständige Ausführen von VVVV zum Testen und Debuggen. Ferner ist es von Vorteil, dass alle Inhalte der In- und Output-Pins auf einen Blick sichtbar sind. Auch Input-Pins die in VVVV nicht sichtbar sind (Kapitel 5.2.3.) können hier dargestellt werden. Dadurch wird das Debugging sehr erleichtert. Der Stand-Alone Host ist zu dem in der Lage, unter Benutzung des in Kapitel 2.2.7. vorgestellten OSC-Protokolls die Daten der Output-Pins an jede beliebige Applikation, die das OSC-Protokoll implementiert, zu senden. Die zweite Komponente der Schnittstelle bildet das Interface, das in C# realisiert ist und von der Zeitleistensteuerung implementiert wird. Durch dieses Interface werden alle Funktionalitäten die zum Entwickeln eines Knotens für den Host VVVV notwendig sind bereitgestellt.

---

[42] Freies Pendant zu .Net, weitere Informationen unter: http://www.mono-project.com

[43] Freies SVN (Subversion) Tool, weitere Informationen unter: http://tortoisesvn.tigris.org

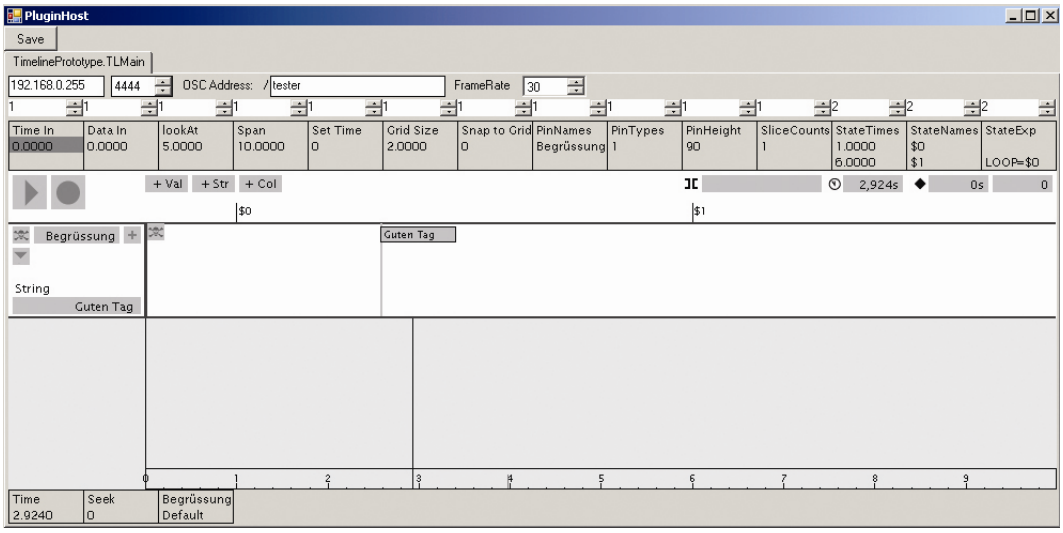


Abb. 69: Knotensimulator als Stand-Alone Applikation

6.2. Implementierung

Die Implementierung des Knotens erfolgt als dynamische Klassenbibliothek, die in einer dll[44] Datei angelegt ist. Durch die Integration der neuen Schnittstelle in VVVV wird ein neues Dateiverzeichnis bereit gestellt, das neben dem Plug-In (in diesem Fall die Zeitleistensteuerung) eine weitere Datei, das Plug-In Interface, enthält. Die Schnittstelle durchsucht nun beim Starten von VVVV diesen Plug-In Ordner und stellt darin gefundene und valide Plug-Ins als Knoten im Knotenmenü bereit.

6.2.1. Feinkonzeption

Bevor auf die Funktionalitäten einzelner Klassen eingegangen wird, werden im Folgenden noch einige Verfahren näher erläutert, die im Systementwurf nur grundlegend spezifiziert wurden.

6.2.1.1. Ablaufsteuerung

Zum Ablaufen und Pausieren des Abspielzeigers werden zwei Uhren benötigt. Sobald die Wiedergabe bzw. Aufnahme gestartet wird, speichert das System die aktuelle Host-Zeit, die durch VVVV übergegeben wird. Anschließend wird diese gespeicherte Zeit von der aktuellen Host-Zeit abgezogen. Dadurch kann die Spielzeit berechnet werden, die seit Betätigung des Abspielknopfes vergangen ist.

[44] Dynamic Link Library

Wird der Abspielknopf erneut betätigt, stoppt die Wiedergabe. Das System befindet sich im Pausenmodus. Eine Berechnung der Pausenzeit wird vorgenommen. Bedient der Benutzer den Abspielknopf erneut, wird die zuletzt berechnete Zeitspanne von der Spielzeit abgezogen, die im Hintergrund weiterberechnet wurde. Damit wird garantiert, dass der Abspielzeiger an der Position fortfährt, an der er pausiert wurde.

6.2.1.2. Automation

Die Automation ermöglicht es, die zuvor beschriebene Spielzeit zu manipulieren. Erreicht der Abspielzeiger die Position eines Zustandes, wird geprüft, ob dieser Zustand eine Sprungfunktionalität enthält. Die Sprungfunktionalität kann zuvor als Statement angegeben werden und wird durch einen regulären Ausdruck (Listing 5) geprüft. Ist das Statement valide, wird ein Input-Pin (*bool*) mit dem Namen des Events erzeugt (Listing 6). Ist dieser Event-Pin *true,* während sich der Abspielzeiger auf der Position des Zustandes befindet, setzt die Automation die aktuelle Abspielzeit auf die Zeit des Sprungziels.

```
pattern = @"^([A-Za-z0-9]+\=+\$+\d{1,3})$";
```

Listing 5: Regulärer Ausdruck zum Überprüfen des Automationsstatements

```
public TLEventPin(IPluginHost Host, string Name)
            {
                  FHost = Host;
                  FEventName = Name;
                  FHost.CreateValuePin(FEventName, true, 1,
                  null, TMPinDirection.cmpdInput,
                  TMSliceMode.cmsmSingle,
                  TMPinVisibility.pivTrue, out FEvent);
                  FEvent._SetSubType(0,1,1,0,false, true,true);
            }
```

Listing 6: Klasse zum anlegen eines Event-Pins

In der derzeitigen Implementierung kann jedem Zustand jeweils nur ein Ereignis zugeordnet werden.

6.2.1.3. Interpolation

Zur Interpolation der Werte zwischen den Keyframes wird immer nur das Keyframe-Paar betrachtet, das sich direkt vor und hinter dem Abspielzeiger befindet. Daher wird garantiert, dass für jedes Keyframe-Paar ein anderer Interpolationsmodus verwendet werden kann.

Linear
Die lineare Interpolation erfolgt auf Grundlage einer einfachen mathematischen Funktion. (Listing 7)

```
double lerp (double t, double x0, double x1){
   return (x0 + t * (x1 - x0));
}
```
Listing 7: Lineare Interpolation

Dabei liegt der Parameter *t* im Intervall [0,1]. Dieser beschreibt die prozentuale Position zwischen den beiden betrachteten Keyframes.

Bézier

Zur Berechnung der Bézier Interpolation zwischen zwei Keyframes ist es notwendig, für einen gegebenen X-Wert einen Y-Wert auf der Kurve zu bestimmen. Die Schwierigkeit besteht jedoch darin, dass sich der X-Wert nicht selbst auf der Kurve befindet, sondern in diesem Fall durch den Abspielzeiger vorgegeben wird. Ein einfaches Auslesen des Wertes mit Hilfe einer Bézier-Formel ist somit nicht möglich. Um den gesuchten Wert zu erhalten, wird ein algebraischer Ansatz verfolgt.

Es wird zunächst für das gegebene X und die X-Koordinaten der vier Kurvenpunkte, eine eindimensionale Bézierfunktion berechnet. Anschließend wird die Umkehrfunktion zum Bézierverfahren gebildet. Dabei kommt ein Algorithmus zum Einsatz der auf den Cardanischen Formeln zur Lösung reduzierter kubischer Gleichungen, beruht [vgl. Führer07, S.5]. Mit dem daraus errechneten Wert *alpha* wird nun eine weitere eindimensionale Bézierfunktion (diesmal mit den Y-Werten der Stützpunkte) aufgerufen. Daraus ergibt sich der gesuchte Y-Wert zur Position des Abspielzeigers.

```
double bezier(double u, double p0, double p1, double p2, double p3) {
  double v = 1-u;
  return p0*v*v*v + p1*3*u*v*v + p2*3*u*u*v + p3*u*u*u;
}
```
Listing 8: Eindimensionale Bézierfunktion

Auf eine Reparametrisierung der Kurve nach Bogenlänge[45] wurde in der Implementierung zunächst verzichtet. Dieses Verfahren eignet sich eher für Keyframe-Animationen deren Grundlage ein Bewegungspfad bildet [vgl. Eberly06A, S.1]. Eine spätere Integration ist jedoch nicht auszuschließen.

Color

In der aktuellen Implementierung erfolgt die Zuweisung einer Farbe zu einem Keyframe durch einen Hexadezimalwert. Das System arbeitet intern jedoch mit dem RGB Farbsystem, da der Pin-Typ *color* einen RGB-Wert benötigt. Es erfolgt zunächst eine Umrechnung der Byte-Darstellung in Dezimalwerte. Anschließend findet eine lineare Interpolation im HSL Farbraum statt. Der RGB-Wert wird dementsprechend in einen HSL-Wert umgerechnet. Dazu wird folgender Algorithmus genutzt:

[45] Numerisches Verfahren, das die berechneten Punkte auf einer Kurve gleichmäßig verteilt und so für eine gleichbleibende Geschwindigkeit beim Ablaufen der Kurve sorgt [vgl. Watt92, S.346].

- *Abbildung*
  Die RGB-Werte werden in einem Intervall [0,1] abgebildet.
- *Maximum & Minimum*
  Der maximale und minimale RGB-Wert wird gesucht.
- *Luminanz berechnen*
- *Farbgleichheit.*
  Gleicht das gefundene Maximum dem Minimum gleichen liegt ein Grauwert vor. Sättigung und Farbton werden auf *0* gesetzt.
  Sind die Werte unterschiedlich, erfolgt eine Untersuchung der Luminanz.
    - *H und S berechnen*
      Der Farbton und die Sättigung werden berechnet
- *Rückrechnung*
  Die Dezimalwerte für Luminanz und Sättigung werden in prozentuale Anteile umgerechnet. Der Farbton wird in einem Winkel des Intervalls [0,360] abgebildet.

Anschließend wird der Farbton zwischen den Keyframes linear interpoliert. Für die Rückrechnung in den RGB-Farbraum existiert ein ähnlicher, jedoch komplexerer Algorithmus.

6.2.1.4. Benutzerschnittstelle

Beschränkung der Bézier Interpolation
Die Bewegungsfreiheit der Tangentenstützpunkte ist im Gegensatz zu einer gängigen Bézierkurve eingeschränkt. Ein Tangentenstützpunkt darf die X-Position eines nachfolgenden bzw. vorangehenden Keyframes nicht überschreiten (Abb. 70, rechts). Dadurch wird garantiert, dass zu jedem X-Wert der Kurve auch nur ein Y-Wert existiert. Mehrdeutigkeiten der expliziten Darstellung werden vermieden [vgl. Encarnacao86, S. 223]. Andernfalls würde das System in einen undefinierten Zustand geraten da nicht bestimmt werden könnte, welcher Y-Wert der Kurve zu betrachten ist (Abb. 70, links).

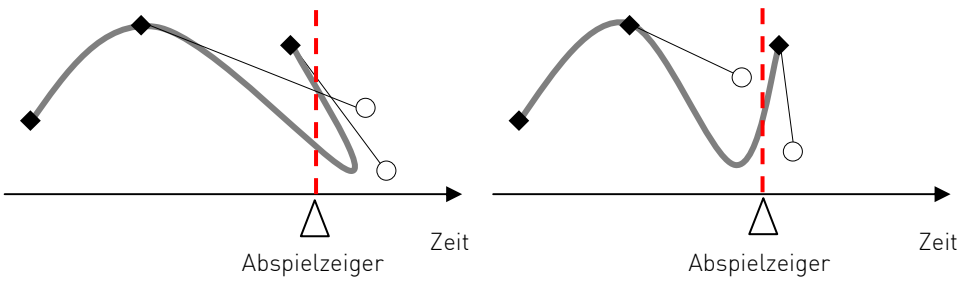


Abb. 70: Beschränkung der Tangentenstützpunkte (rechts)

Sicht auf den Zeitstrahl

Da der Zeitstrahl durch *min.double* und *max.double* begrenzt ist, aber nicht der gesamte Zeitstrahl angezeigt werden soll, muss eine Sicht definiert werden die einen Ausschnitt des Zeitstrahls beschreibt. Es wird ein Punkt *lookAt* definiert, der angibt, welche Zeit des Zeitstrahls von Interesse ist. Dieser Punkt wird zentriert im Editorfenster angezeigt. Zusätzlich existiert ein weiterer Parameter *span* mit dem sich ein Bereich definieren lässt, der vor und nach dem Punkt *lookAt* betrachtet wird (Abb. 71).

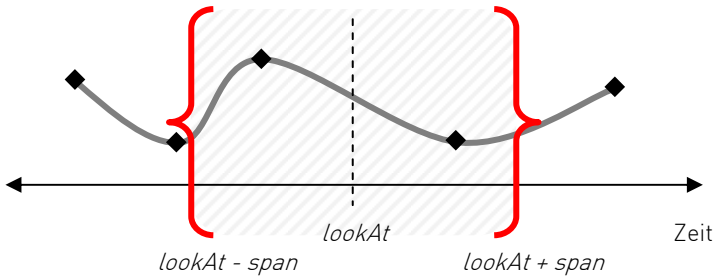


Abb. 71: Sicht auf den Zeitstrahl

Wird der Parameter Span verändert, so lässt sich der zu betrachtende Bereich vergrößern bzw. verkleinern. Beide Parameter können sowohl mit der Maus als auch von außen durch Input-Pins verändert werden.

Slices

Ein ähnliches Prinzip ist bei der Betrachtung des Wertebereichs der Keyframes des Typs *value* implementiert. Hier lässt sich der zu betrachtende Wertebereich des gesamten Output-Pins definieren. Dadurch wird eine Sicht auf der Y-Achse erzeugt.

Jeder der angelegten Slices bildet diesen Wertebereich vollständig ab (Abb. 72).

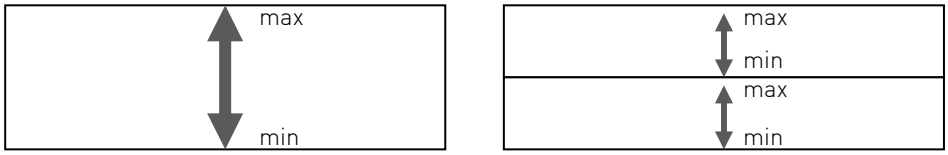


Abb. 72: Wertebereiche der Slices (rechts ein Slice, links mehrere Slices)

Grafische Implementierung

Zum Zeichnen der Benutzeroberfläche kommen ausschließlich GDI+[46] Funktionen zum Einsatz. Weiterhin wurde bei allen Zeichenfunktionen darauf geachtet, diese austauschbar zu gestalten. So ist es ohne größere Schwierigkeiten möglich, die GDI+ Schnittstelle z.B. durch Cairo[47] zu ersetzen. Da die Funktionalität der Anwendung im Vordergrund der Arbeit stand, ist zunächst nur eine einfache grafische Benutzeroberfläche implementiert. Das finale Erscheinungsbild des

[46] Zeichen-API der Firma Microsoft.

[47] Freie Zeichen-API. Weitere Informationen: http://cairographics.org

Entwurfs (Kapitel 5.3.) wird im Rahmen dieser Arbeit nicht eingebunden. Allerdings bedeutet das keine funktionalen Einschränkungen und ist somit eher eine Frage der Ästhetik.

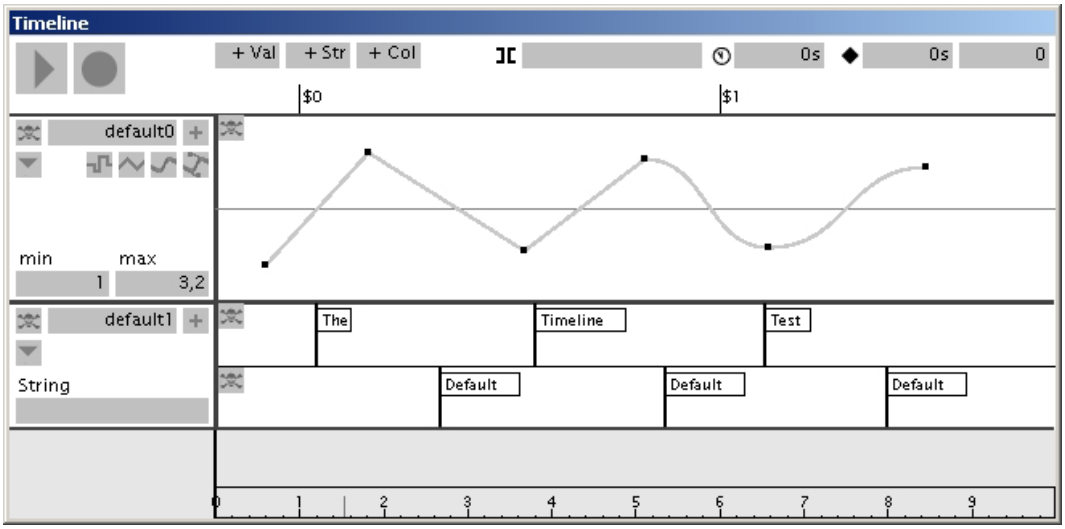


Abb. 73: 1. Datenspur (*value*) kombiniert Interpolationsverfahren. 2. Datenspur (*string*) beinhaltet mehrere Slices

Funktionen

Abbildung 74 zeigt verschiedene Editierfunktionen für Keyframes. Nähert sich der Benutzer einem Keyframe, wird dieses vergrößert dargestellt (Abb. 74, links, mitte). Dadurch wird ein sehr einfaches visuelles Feedback realisiert [vgl. Tognazzini93, S.139]. Um mehrere Keyframes zu selektieren, kann der Benutzer einen Rahmen um eine gewünschte Gruppe ziehen. Diese Gruppe wird visuell markiert um eine Unterscheidung zu verdeutlichen [vgl. Raskin01, S.129]. Anschließend kann für die selektierte Gruppe ein Interpolationsmodus gewählt werden (Abb. 74, rechts). Für einzeln selektierte Keyframes kann eine genaue Positionierung in Zeit und Wert erfolgen (Abb.75, links). Sind mehrere Slices in einer Datenspur vorhanden, lassen sich die Interpolationskurven gemeinsam betrachten. Wird in dieser Ansicht ein Keyframe erstellt, wirkt sich dies auf alle betrachteten Kurven aus (Abb. 75, mitte). Beim Selektieren eines Zustandes, wird das aktuell zugewiesene Ereignis angezeigt und kann modifiziert werden (Abb. 75, rechts).

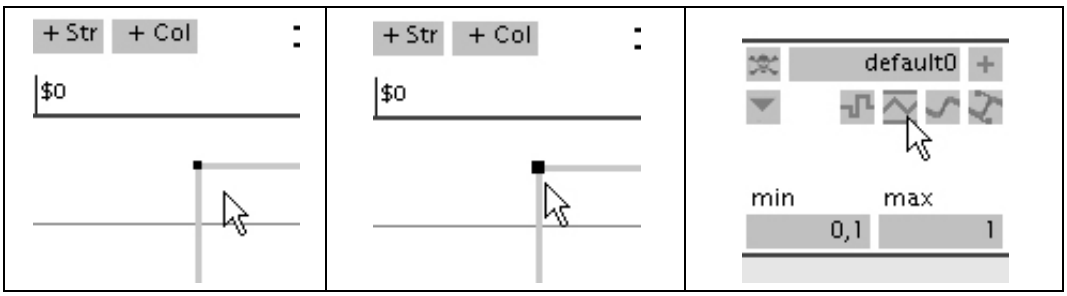


Abb. 74: Editierfunktionen

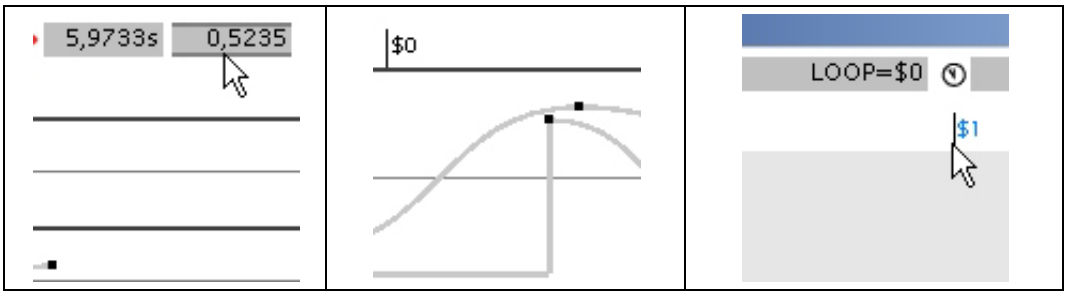


Abb. 75: Editierfunktionen

6.2.2. Details der Implementierung

Inititalisierung
Legt der Benutzer den Knoten an, wird zunächst eine Instanz erzeugt. Den Einstiegspunkt der Applikation bildet die Klasse *TLMain*, welche auch das zuvor erläuterte Interface implementiert. Durch die Instanziierung der Interfacekomponente *IPluginHost* kann in der Anwendung direkt auf die Schnittstelle zugegriffen werden.

Der Mainloop[48] der Applikation wird durch das Interface aufgerufen. Die Methode *Evaluate* der Klasse *TLMain* wird einmal pro VVV Berechnungsframe (Kapitel 2.2.8.) aufgerufen. Wird die Framerate der Berechnungen in VVV manuell ausgebremst[49], wirkt sich dies auch auf die Zeitleistensteuerung aus, da die Methode *Evaluate* in einer geringeren Frequenz aufgerufen wird.

6.2.2.1. Pins

Die Implementierung des Interface ermöglicht das Anlegen von In- oder Output-Pins. Bevor ein Pin angelegt werden kann muss zunächst sein Datentyp spezifiziert werden. Anschließend bietet die Methode *CreateValuePin* verschiedene Eigenschaften einen Pin zu spezifizieren (Listing 9).

```
void CreateValuePin(string Name, bool Diff, int Dimension,
                    string[] DimensionNames, TMPinDirection PinDirection,
                    TMSliceMode SliceMode, TMPinVisibility Visibility,
                    out IValueIO Pin);
```

Listing 9: Methode zum Erstellen eines Value-Pins

Statische Pins
Abbildung 76 zeigt die zur Objektinitialisierung angelegten statischen Pins.

[48] Funktion die zyklisch aufgerufen wird.
[49] Der Knoten *MainLoop* erlaubt die manuelle Festlegung der Berechnungsframerate.

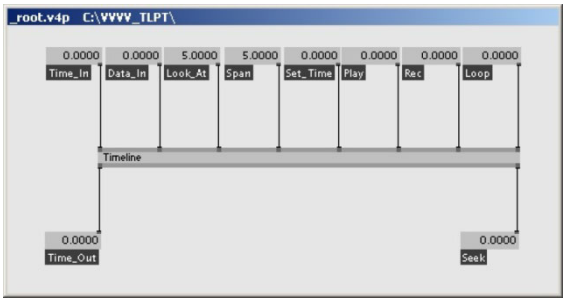


Abb. 76: Timeline Knoten mit statisch angelegten Pins

- *Time_In*
  Erlaubt die Eingabe einer Zeit in das System.
- *Data_In*
  Durch diesen Pin können Werte zu Keyframe-Generierung eingegeben werden.
- *Look_At*
  Bestimmt den Punkt des Zeitstrahls der zentriert im Viewport liegt.
- *Span*
  Durch den Pin Span kann der Bereich des Zeitstrahls angegeben werden, der um den Punkt Look_At betrachtet werden soll.
- *Set_Time*
  Schaltet zwischen interner und externer Zeitgebung um.
- *Play*
  Startet bzw. pausiert das Ablaufen der Zeitleiste.
- *Record*
  Startet bzw. pausiert das Aufzeichnen von Werten, die am Pin *Data_In* anliegen.
- *Loop*
  Ermöglicht das Setzen des Wiederholungsereignisses.
- *Time_Out*
  Gibt die aktuelle Position des Abspielzeigers aus.
- *Seek*
  Schaltet für einen Berechnungsframe von *0* nach *1* sobald ein Ereignis der Automation ausgelöst wird.

Dynamische Pins

Dynamische Pins werden erstellt, sobald eine Datenspur erzeugt wird. Dies ist der Fall, wenn der Benutzer manuell den Befehl dazu erteilt oder Daten geladen werden. Dynamische Pins dienen zur Ausgabe der Werte der Datenspuren oder zum Speichern von Einstellungen. Weiterhin erfolgt eine Einteilung in global- und lokal-dynamische Pins. Global-dynamische Pins werden in der Klasse *TLMain* zu Initialisierung der ersten Datenspur erzeugt und sind unabhängig vom Typ der Spur. Lokal-dynamische Pins werden dagegen in den abgeleiteten Klassen der einzelnen Datenspuren erzeugt, da diese typabhängig sind. Zu den lokalen Pins gehört beispielsweise auch der Output-Pin einer Datenspur. Für Spezialfälle

existieren Pins die einen zweidimensionalen Vektor speichern können. Die Dimension wird bei der Instanziierung des Pins angegeben.

Die Speicherung der Daten erfolgt dabei durch VVVV in einem spezifizierten XML[50]-Format. Nach der Instanziierung des Knotens werden zunächst alle Pins sequentiell angelegt die in dieser Datei gespeichert vorliegen. Sind alle benötigten Pins vorhanden werden aus den enthaltenen Daten entsprechende Datenspuren mit Keyframe-Daten erzeugt. Dies geschieht durch eine *PinChanged*-Methode, die vom Interface dann aufgerufen wird, wenn sich ein Wert innerhalb eines Pins ändert.

Löscht der Benutzer Keyframes oder eine komplette Datenspur, werden die zugeordneten Einträge aus den globalen Pins gelöscht. Danach ist eine Sortierung der verbliebenen Einträge erforderlich. Diese erfolgt anhand der Pin-Namen, die eindeutig sind, sowie der Indizierung der Datenspuren die in einer streng typisierten Liste gespeichert sind. Lokale Pins werden durch den Aufruf einer *Dispose*-Methode der jeweiligen Datenspur zerstört (Listing 10).

```
public override void Dispose(){
	// SLICE DISPOSE CALL
	//////////////////////
	for (int i=0; i<OutputSlices.Count; i++){
		OutputSlices[i].Dispose();
	}

	// CLEAR & DELETE SLICES
	//////////////////////
	OutputSlices.Clear();

	// DELETE PINS
	//////////////////////
	host.DeletePin(FArm);
	FArm = null;

	host.DeletePin(FVOut);
	FVOut = null;

	base.Dispose();
}
```

Listing 10: Dispose-Methode einer Datenspur

6.2.2.2. Klassen

Abschließend werden ausgewählte Klassen der Implementierung näher beschrieben.

[50] Extensible Markup Language, ein Standard zur Modellierung von Daten

TLPinManager
Von dieser abstrakten Klasse werden die Datenspuren abgeleitet. In ihr existierten Methoden, mit denen die Werte der Datenspuren in die statischen Hidden-Pins des Systems und somit beim Speichern des gesamten VVV-Patches in eine XML-Datei geschrieben werden. Die globalen Eigenschaften einer Datenspur sind: Name, Höhe und Anzahl der Slices.

TLPinManagerValue
Diese Klasse generiert eine Datenspur vom Typ *value*. Bei der Instanziierung wird ein Output-Pin vom Typ *value* erstellt und ein Slice sowohl visuell, als auch im Output-Pin angelegt. Der Output-Pin erhält einen eindeutigen Namen des Typs *Default + Index*.

TLPinManagerString
Die Klasse generiert eine Datenspur vom Typ *string*. Auch sie legt einen typisierten Output-Pin an, allerdings vom Typ *string*. Es existieren zusätzlich zwei Hidden-Pins die Zeit und Wert der Keyframes speichern.

TLPinManagerColor
Die Klasse generiert eine Datenspur vom Typ *color*. Es wird ein Output-Pin vom Typ *color* sowie ein Slice generiert. Zur Speicherung der Keyframe-Daten werden ein zweidimensionaler Hidden-Pin vom Typ *value*, sowie ein Hidden-Pin des Typs *color* angelegt.

TLSliceManager
Mit dieser abstrakten Klasse wird ein Slice in einem Pin abgebildet. In ihr werden die Keyframe-Objekte verwaltet und die Interpolationen anhand der Position des Abspielzeigers ausgeführt. Diese Modularisierung erlaubt das spätere Hinzufügen neuer Datentypen. Zudem stehen hier verschiedene virtuelle Methoden bereit, die von den abgeleiteten Klassen überschrieben werden. Die wichtigsten sind:

- *Evaluate*
  wird zyklisch aufgerufen, sobald die Abspielsteuerung aktiv ist
- *AddKeyframe*
  erzeugt ein neues Keyframe
- *DeleteKeyframe*
  löscht ein bestehendes Keyframe
- *SortKeyframes*
  sortiert die Keyframes nach ihren Zeiten

Die Slice-Klassen für die jeweiligen Datentypen werden vom *TLSliceManager* abgeleitet.

TLGraphicLib
Da zum Zeichnen keine vorgefertigten .Net Elemente benutzt werden aber gleichzeitig oft benutzte Komponenten nicht in jeder Klasse neu instanziiert werden sollen, beinhaltet diese Klasse grafische Elemente und kann bei Bedarf instanziiert werden.

7. Test

Dieses Kapitel gewährt einen kleinen Überblick über Einsatzmöglichkeiten des Systems. Da die Szenarien in Verbindung mit VVV extrem vielfältig sind, geschieht dies eher exemplarisch als konkret, um ein möglichst breites Spektrum aufzeigen zu können.

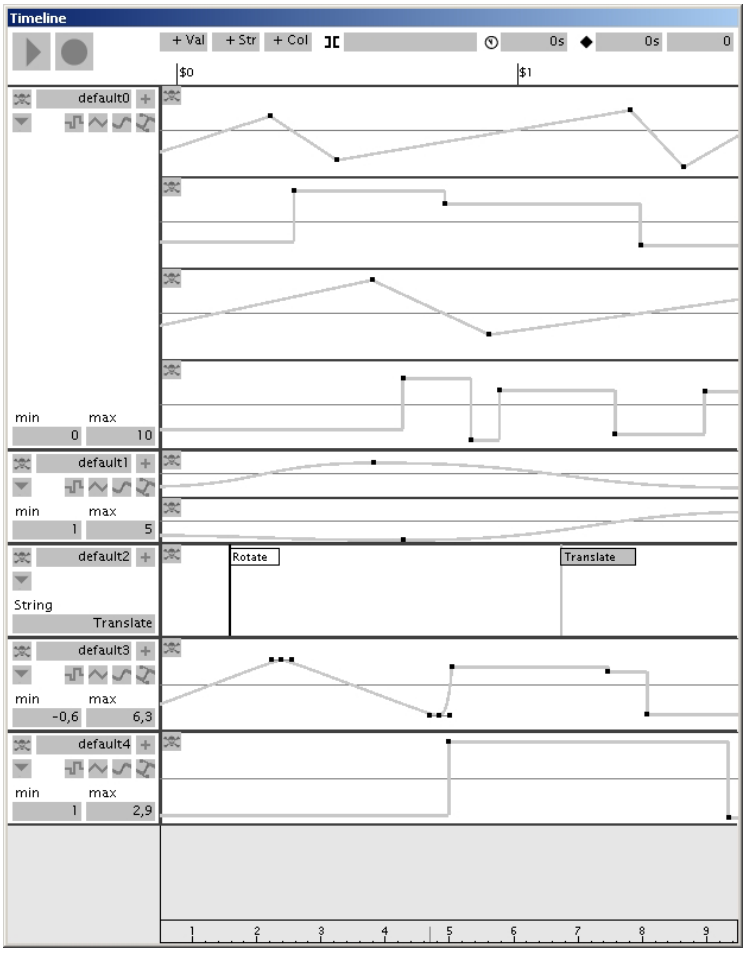
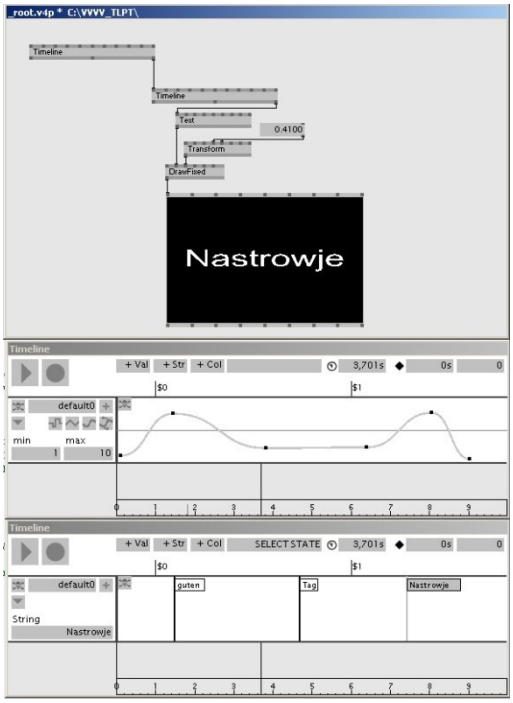
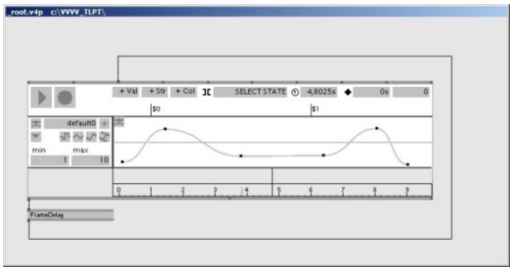


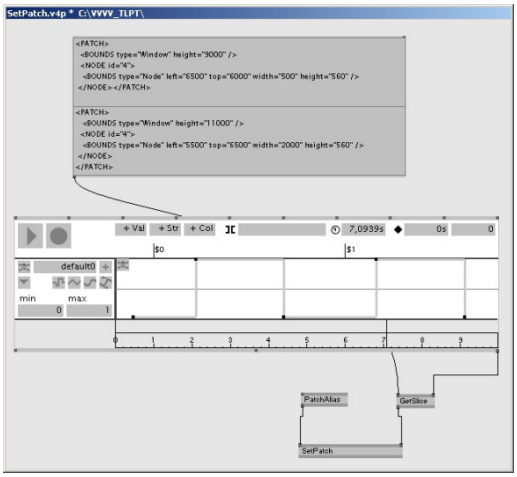
Abb. 77: Grafischer Editor der Zeitleistensteuerung



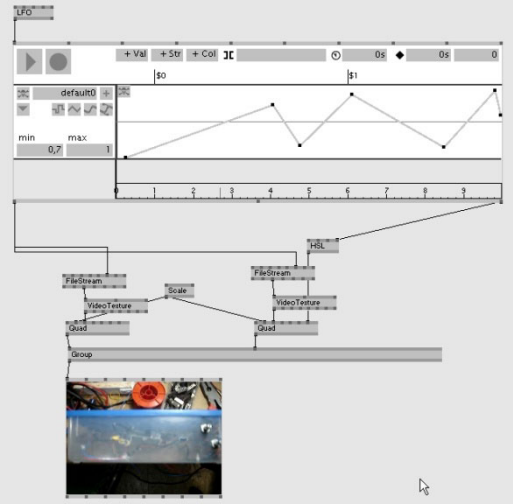
BEISPIEL01
Hierarchische Anordnung von zwei Zeitleisten. Die obere Zeitleiste steuert mit der Interpolationskurve den zeitlichen Ablauf der unteren Zeitleiste. Diese gibt zu spezifischen Zeiten Strings zurück, die über einen Direct-X Renderer ausgegeben werden.



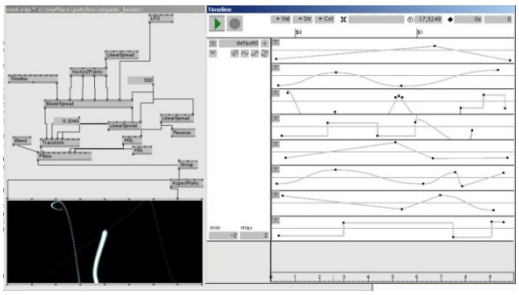
BEISPIEL02
Durch den Einsatz des Knoten *FrameDelay* wird der Output-Pin, der die aktuelle Position des Abspielzeigers ausgibt, mit dem Input-Pin verbunden der bestimmt welcher Zeitpunkt betrachtet werden soll. Wird die Zeitleiste nun abgespielt, läuft die Ansicht immer mit der Position des Abspielzeigers mit.



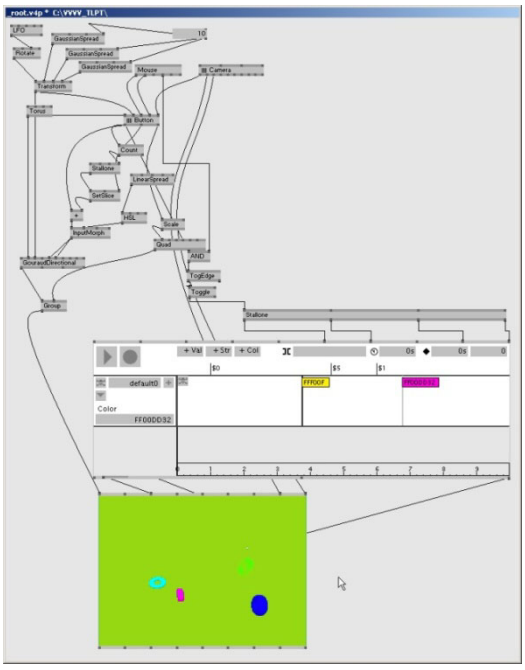
BEISPIEL03
In Kombination mit dem Knoten *SetPatch* wird die Zeitleistensteuerung zu einem komplexen Werkzeug. Der Knoten *SetPatch* erlaubt es andere Knoten durch XML Statements zu erstellen. Dies kann jetzt zeitgesteuert geschehen. So lässt sich das System selbst zeitabhängig modifizieren. Die Zeitleiste befindet sich in der Patch-Ansicht. Der grafische Editor ist direkt in das Patchfenster integriert.



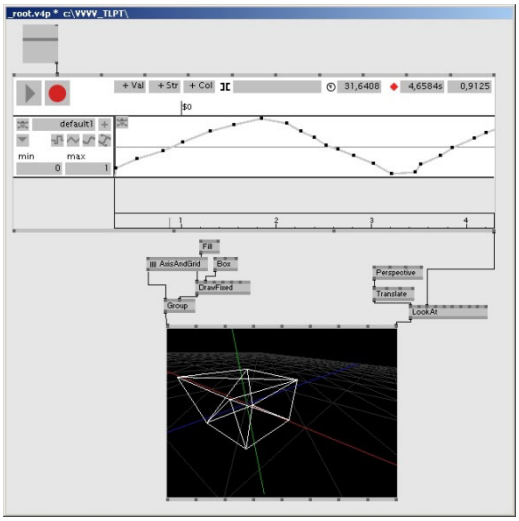
BEISPIEL04
Die Abspielzeit der Zeitleiste wird durch den Knoten *LFO* gesteuert. Die Zeitleiste spielt zwei Videodateien ab und überblendet zwischen diesen.



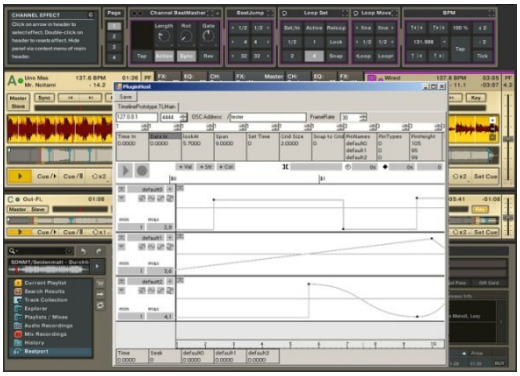
BEISPIEL05
In diesem Beispiel steuern die Datenspuren der Zeitleistensteuerung das Verhalten eines Partikelsystems. Dadurch werden Geschwindigkeit und Positionen der Partikel beeinflusst.



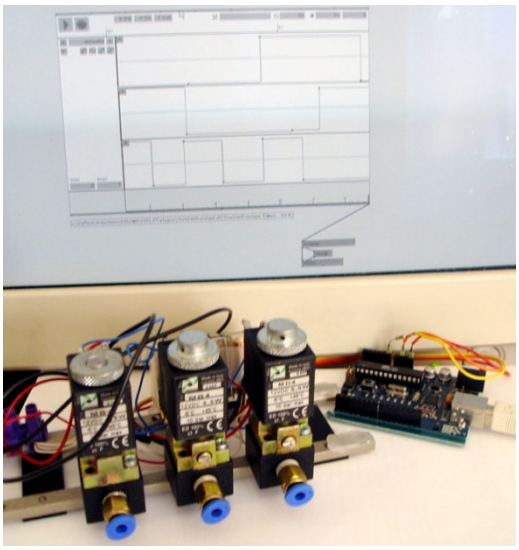
BEISPIEL06
Vier Bereiche sind durch die Automation erstellt worden. Durch das Anwählen eines der 3D-Objekte wird ein Ereignis ausgelöst und die Zeitleiste springt in einen Bereich. Daraufhin wird die Hintergrundfarbe des Renderers durch eine Datenspur vom Typ *color* geändert. Zeitgleich könnten natürlich auch viele weitere Parameter der 3D-Szene geändert werden. Diese Funktionalität ermöglicht es, Menüstrukturen mit Hilfe der Zeitleistensteuerung abzubilden.



BEISPIEL07
An den Input-Pin *Data-In* ist ein Schieberegler angeschlossen. Die Werte dieses Reglers werden bei Veränderung in die Datenspur aufgezeichnet und können dann beliebig abgespielt werden. In diesem Beispiel wird die Y-Translation einer Kamera in einem dreidimensionalen Raum reguliert.



BEISPIEL08
Die Zeitleistensteuerung wird im Stand-Alone Modus ausgeführt. Im Hintergrund läuft die Anwendung Traktor 3 von Native Instruments (eine Audiosoftware die es ermöglicht die Geschwindigkeit von MP3´s aneinander anzupassen und die Audiodateien zu mischen). Die Zeitleisteuerung kommuniziert über das OSC-Protokoll mit Traktor und kann so beliebige Parameter steuern. (in diesem Fall die Lautstärke)



BEISPIEL09
Dieses Beispiel demonstriert den Einsatz des Knotens zur Steuerung von externer Hardware. Angesteuert werden drei magnetische Ventile, mit denen der Zufluss von Luft oder Wasser reguliert werden kann. Durch die integrierte Zeitleistensteuerung ist es möglich die Ventile zeitgesteuert schalten zu lassen. Dazu werden in VVVV nur noch vier Knoten benötigt. Für die Kommunikation zwischen Rechner und Hardware wird in diesem Beispiel das Arduino[51] IO-Interface benutzt, welches über die serielle Schnittstelle angesteuert wird.

Abb. 78: Anwendungsbeispiele

[51] Hardwareschnittstelle, weitere Informationen unter: http://www.arduino.cc

8. Fazit und Ausblick

Ziel dieser Arbeit war, zu untersuchen, inwieweit sich der Echtzeitansatz der grafischen Programmiersprache VVVV mit einer linearen Zeitleistensteuerung kombinieren lässt und welche Vorteile sowie neue Bearbeitungsmethoden daraus entstehen können.

Dazu mussten zunächst die Schwachstellen der Bearbeitung zeitgesteuerter Parameter in VVVV aufgedeckt werden. Anhand verschiedener Beispiele wurde deutlich, wie wichtig eine Überarbeitung der Strukturen war. Um näher auf die Gegebenheiten zeitleistenbasierter Systeme eingehen- und somit klare Anforderungen an ein neues System stellen zu können, fand eine Betrachtung bereits vorhandener Lösungen statt. Während der Entwurfsphase des Systems wurden anschließend sowohl Ansätze herkömmlicher Animationssysteme aufgegriffen als auch explizit neue Wege gesucht, die es ermöglichen sollen, die Zeitleistensteuerung flexibel und losgelöst von der klassischen Animation einzusetzen. Als Beispiel dafür ist die Integration verschiedener Datentypen zu nennen. Eine Kernanforderung bildete die Eingliederung in die Programmierumgebung von VVVV. Durch die Implementierung als Knoten ist eine vollständige Integration der Zeitleistensteuerung in den Datengraph von VVVV gegeben. Durch die Kombination von Echtzeitsystem und linearem Medium ergeben sich viele neue Bearbeitungsmethoden. So lässt sich die Zeitleistensteuerung sowohl grafisch bedienen, wie es in herkömmlichen Systemen üblich ist, als auch durch andere Knoten fernsteuern. Auch die Möglichkeit, mehrere Zeitleisten miteinander zu kombinieren, ist in herkömmlichen Systemen nicht gegeben.

Wird der Zustand vor der Implementierung betrachtet, kann festgestellt werden, dass die zeitgesteuerte Parameterbearbeitung in VVVV massiv vereinfacht wurde. Die Anzahl der benötigten Knoten reduziert sich drastisch. Da die komplette Funktionalität, die zuvor für jedes Projekt mühsam neu programmiert werden musste nun auf Knopfdruck verfügbar ist, bringt das eine sehr hohe Zeitersparnis mit sich. Zusätzlich wird das Einarbeiten in die Programme eines anderen Mitarbeiters deutlich erleichtert, da nun nur noch ein Knoten beherrscht werden muss. Ein weiterer positiver Aspekt ist die Übersichtlichkeit und Platzersparnis in einem Patch. Durch die Integration der Automationsfunktionalität wird auf

längere Sicht der *Automata*-Knoten vollständig abgelöst werden. Es wurde ein Grundkonzept präsentiert mit dem sich ein Automat zumindest teilweise grafisch darstellen lässt. Der zusätzliche Faktor Zeit, über den ein endlicher Automat normalerweise nicht verfügt, verspricht neue Anwendungsgebiete. Die nötige textuelle Eingabe konnte auf ein 2-Tupel[52] beschränkt werden.

Bei der Implementierung der Anwendung wurde auf Modularität geachtet. Es ist ohne weiteres möglich, das Programm um zusätzliche Funktionen zu erweitern. Durch den Einsatz von Mono ist es möglich, das System auch unter Linux und Mac-OS zu benutzen. Die prototypische Implementierung bildet eine Basis für die weitere Entwicklung. Das entstandene System ist zwar funktionstüchtig, aber die Tür zu einem professionellen Zeitleisteneditor konnte gerade einmal aufgestoßen werden. Es gilt zunächst, die Editierfunktionalitäten in realen Projekten, z.B. bei Meso, auf Robustheit und Flexibilität zu testen und ggf. zu optimieren. Gerade die Perfektionierung einer komplexen Benutzerschnittstelle benötigt viel Zeit. Dazu wird das System bald Teil einer offiziellen Release-Version von VVVV. Durch die große- und ausprobierfreudige Community ist für reichlich Feedback gesorgt. Ideen für weitere Entwicklungen existieren aber bereits jetzt. Eine der prägnantesten ist die Implementierung eines komplett nicht-linearen Modus. Dieser soll es ermöglichen, erstellte Kurvenverläufe als Objekte zu betrachten und diese in einer speziellen Ansicht miteinander kombinieren, vervielfältigen, zeitlich strecken oder stauchen zu können. Auch die Integration eines Stiftwerkzeuges zur direkten Manipulation von Interpolationsverläufen ist angedacht.

In Kapitel 7 wurde ein Einblick in die Verwendungsmöglichkeiten des Systems vermittelt. Wie vielfältig die letztendlichen Einsatzszenarien sind, lässt sich im Moment nur schwer abschätzen. Insbesondere, da die Zeitleistensteuerung kein abgeschlossenes System ist, sondern nur Teil einer komplexen Programmiersprache.

[52] n-Tupel: Geordnete Zusammenstellung von Objekten.

Anhang A. Quellcodeverzeichnis

Anhang B. Abbildungsverzeichnis

Anhang C. Tabellenverzeichnis

Anhang D. Literaturverzeichnis

Ackerman82	William B. Ackerman, Data Flow Languages, Computer Vol. 15, No. 2, 1982
Answ07	http://www.answers.com/topic/b-zier-curve-2, (Stand 03.2007)
Armstrong05	Jim Armstrong, Hermite curves, F/X Technote 2112 No.: TN-05-002, November 2005
Benes01	Bedrich Benes, Animation Curves, Purdue University, 2001
Burnett94	Margeret M. Burnett, Marla J. Baker, A Classification System for Visual Programming, Journal of Visual Languages and Computing Nr. 5, Academic Press, 1994
Burnett99	Margeret M. Burnett, Visual Programming, In John G. Webster, Encyclopidia of Electrical and Electronics Engineering, John Wiley and Sons Inc., New York, 1999
CL07	http://www.cl.unizh.ch/siclemat/lehre/ss06/mul/script/html/scriptse2.html, (Stand 03.2007)
Cubic07	http://cubic.org/docs/hermite.htm, (Stand 03.2007)
Cubic07A	http://www.cubic.org/docs/bezier.htm, (Stand 03.2007)
Dießl01	Jörg Dießl, Gestaltung und Implementierung einer grafischen Programmiersprache zur Echtzeitvideosynthese
DMA07	http://www.dma.ufg.ac.at/app/link/Grundlagen%3A3D-Grafik/module/14155, (Stand 03.2007)
DR07	http://dr13.de, (Stand 04.2007)
Drayton03	Peter Drayton, Ben Albahari, Ted Neward, C# IN A NUTSHELL, O´Reilly, 2003
Eberly06	David Eberly, Kochanek-Bartels Cubic Splines, Geometric Tools Inc., 2006
Eberly06A	David Eberly, Moving at Constant Speed, Geometric Tools Inc., 2006
Encarnacao86	J. Encarnacao, W. Straßer, Computer Graphics, Oldenbourg 1986
Führer07	Prof. Dr. Lutz Führer, Kubische Gleichungen und die widerwillige Entdeckung der komplexen Zahlen, Institut für Didaktik der Mathematik der Johann Wolfgang Goethe-

	Universität, Frankfurt am Main, http://www.math.uni-frankfurt.de/~fuehrer/forschung/Cardano.pdf, [Stand 04.2007]
Günther01	Ronny Günther, Bézierkurven, Marie-Curie-Gymnasium, 2001
Hausig98	Nils Hausig, Entwurf und Implementierung eines Werkzeugs zur Generierung von Animationen durch automatische Navigation in schlauchförmigen Hohlstrukturen, Universität Hamburg, 1998
Kling02	Garry Kling, Andreas Schlegel, OSCar and OSC: Implementation and use of Distributed Multimedia in the Media Arts, ACM-Press, 2002
Max06	Max/MSP Complete Documentation, http://www.cycling74.com/download/maxmsp46doc.zip, (Stand 04.2007)
Mutafchiev07	Stoyan Mutafchiev, Endliche Automaten, n.A. http://www.ps.uni-sb.de/courses/seminar-ss03/XML/Stoyan(Ausarbeitung).pdf (Stand 03.2007)
Noack n.A.	Hartmut Noack, Freiformkurven und –flächen in CAD-Systemen, Fachhochschule Hamburg, n.A.
Oliver94	Dick Oliver, Graphikzauber, IWT 1994
Pipenbrink98	Nils Pipenbrink, Hermite Curve Interpolation, Cubic, 1998
Polevoi99	Rob Polevoi, 3D Studio Max R3 In Depth, Coriolis, 1999
Poswig96	Jörg Poswig, Visuelle Programmierung, Hanser 1996
Raskin01	Jef Raskin, Das intelligente Interface, Addison-Wesley, 2001
Schiffer05	http://www.swe.uni-linz.ac.at/people/schiffer/se-96-19/se-96-19.htm, (Stand 03.2007)
Schiffer98	Stefan Schiffer, Visuelle Programmierung, Addison-Wesley, 1998
Schürr01	Prof. Dr. Andy Schürr, Folien zur Vorlesung „Visuelle Programmierung – Programming Languages and Environments“, Universität der Bundeswehr, München 2001
Shu88	Nan C. Shu, Visual Programming Languages, Van Nostrand Reinhold, 1988
TCP05	n.A., Information Sciences Institute, University of Southern California, Transmission Control Protocol, RFC 793, 1981 http://tools.ietf.org/html/rfc793 (Stand 03.2007)
Thalmann n.A.	Nadia Magnenat Thalmann, Daniel Thalmann, Computer Animation in Future Technologies, MIRA Lab – University of Geneva, Computer Graphics Lab – Swiss Federal Institue of technology, n.A.
Tognazzini93	Bruce Tognazzini, TOG on Interfaces, Apple Computers Inc., 1993
Tufte01	Eward R. Tufte, The Visual Display of Quantitive Infomation, Graphic Press, 2001
UDP05	J. Postel, User Datagram Protocol, RFC 768, 1980

http://tools.ietf.org/html/rfc768 (Stand 03.2007)

VVVV07 Webseite der Programmierumgebung VVVV, http://vvvv.org, (Stand 04.2007)

VVVV07A Webseite der Programmierumgebung VVVV, http://vvvv.org/tiki-index.php?page=Creating+Feedback+Loops, (Stand 04.2007)

VVVV07B Webseite der Programmierumgebung VVVV, http://vvvv.org/tiki-index.php?page=Basic+Programming+Concepts, (Stand 04.2007)

VVVV07C Webseite der Programmierumgebung VVVV, http://vvvv.org/tiki-index.php?page=Tutorial+IOBoxes, (Stand 04.2007)

Watt98 Alan Watt, Mark Watt, Advanced Animation and Renderung Techniques, Addison-Wesley, 1998

Watt02 Alan Watt, 3D-Computergrafik, Addison-Wesley, 2002

Wright02 Max Wright, Open Sound Control Specification Version 1.0, 2002 http://www.cnmat.berkeley.edu/OpenSoundControl/OSC-spec.html, (Stand 03.2007)

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Ingolf Heinsch

Frankfurt am Main, 10. April 2007