# Visual Programming im Überblick

Philipp Merkel*
Seminar Perzeption und Interaktive Technologien
Universität Ulm

## Zusammenfassung

Die Idee der visuellen Programmierung ist fast so alt wie die Programmierung selbst. Diese Ausarbeitung definiert den Begriff des Visual Programming und grenzt ihn von verwandten Konzepten ab. Es werden verschiedene Formen visueller Programmiersprachen vorgestellt, gemeinsame Eigenschaften benannt, Probleme sowie Vor- und Nachteile visueller Programmierung untersucht und daraus resultierende Anwendungsgebiete abgeleitet. Es ergibt sich, dass visuelle Programmierung insbesondere bei Programmiersprachen für spezielle Einsatzgebiete und mit Programmierparadigmen, die sich verbal schlecht umsetzen lassen, geeignet ist.

## 1 Einleitung und Motivation

Seit es programmierbare Rechenmaschinen gibt, werden die zu ihrer Programmierung dienenden Sprachen immer mehr der menschlichen Denkweise angepasst. Von Lochkarten über Assembler, imperative Hochsprachen wie Pascal oder C bis hin zu objektorientierten, funktionalen oder logischen Programmiersprachen wurden immer neue Wege gefunden, die Programmierung weniger nach der technischen Arbeitsweise der Maschine, sondern zunehmend nach der Art des zu lösenden Problems zu richten.

Doch auch die Mehrzahl dieser heute verwendeten Programmiersprachen bedient sich immer noch einer klassischen Reintext-Notation des Programms. Da Menschen jedoch nicht nur verbal, sondern auch in zwei- und dreidimensionalen Bildern und Szenerien denken, liegt es nahe, auch grafische Objekte zur Programmierung heranzuziehen und ganze Programme als visuelle Kompositionen solcher Objekte zu erstellen.

Tatsächlich gibt es solche Versuche schon fast so lange, wie es textuelle Programmiersprachen gibt. Unter dem Oberbegriff „Visual Programming“ wurden zahlreiche Ansätze entwickelt, über die dieser Text einen Überblick geben möchte. Außerdem wird er ihre Vor- und Nachteile, Möglichkeiten und Grenzen untersuchen.

## 2 Grundlagen

### 2.1 Definition

Visuelle Programmierung (Visual Programming, VP) ist im allgemeinen Sinn eine Programmierung unter Verwendung visueller, grafischer Werkzeuge.

Im engeren Sinne bedeutet bedeutet VP, dass ein Programm in einer visuellen Programmiersprache (Visual Programming Language, VPL) beschrieben wird. Eine solche VPL ist eine Programmiersprache, bei der der Quellcode eines mit ihr erstellten Programms aus grafischen Elementen besteht. Zudem wird mehr als eine Dimension verwendet, um die Semantik des Programms festzulegen (vgl. [Burnett 1999]).

Bei klassischen, textuellen Programmiersprachen ist dies nicht der Fall: Der Text kann zwar in modernen Editoren mit visuellen Hilfsmitteln wie Syntax Highlighting oder Code Folding erstellt und betrachtet werden, diese visuelle Darstellung ändert jedoch nichts an der Semantik des Codes. Auch kann der Code mittels Zeilenumbrüchen, Kommentaren und Einrückungen vom Programmierer visuell gegliedert werden, jedoch sind dies nichts als Hilfsmittel für den Entwickler, die vom Compiler oder Interpreter verworfen werden.

Eine VPL hingegen verwendet grafische Elemente für die Programmierung. Die Bestandteile eines Programms werden hier durch grafische Formen, Symbole und Text repräsentiert, die auf einer zwei- oder auch mehrdimensionalen Fläche angeordnet werden. Meist werden diese grafischen Objekte nicht wie bei verbalen Sprachen sequentiell hintereinander gesetzt, sondern können mehr oder weniger frei positioniert werden. Je nach Positionierung der einzelnen Elemente, auch im Verhältnis zur Position der anderen Programmbestandteile, kann sich die Semantik des Programms ändern. Häufig wird dies auch durch Verbindung der Bestandteile mit Linien oder Pfeilen erreicht.

Der Programmiervorgang erfolgt nicht durch Eingeben des Programmcodes über die Tastatur, sondern durch Komposition der visuellen Objekte mit Hilfe direkter Manipulation. Da praktisch jede VPL eine eigene Programmierumgebung mitbringt, eine so genannte VPE (Visual Programming Environment), sollen Sprache und Umgebung im Folgenden stets gemeinsam betrachtet werden.

### 2.2 Abgrenzung

Neben VPLs gibt es auch weitere Programmiersysteme, die manchmal unter dem Oberbegriff Visual Programming betrachtet werden. So werden Entwicklungsumgebungen, bei denen die Programmierung mit klassischen, rein textuellen Sprachen erfolgt, die jedoch auch visuelle Tools für Oberflächengestaltung, Debugging, Erkennung von Abhängigkeiten o.ä. enthalten, ebenfalls oft als VPEs bezeichnet. Hierzu zählen insbesondere die Produkte der „Visual Studio“-Reihe von Microsoft. Hierbei sind jedoch wirklich nur die Entwicklungsumgebungen visuell, die Programmiersprachen an sich nicht (vgl. [Schiffer 1996]). Daher werden diese im Folgenden auch nicht weiter betrachtet werden.

### 2.3 Ziele

Eines der wichtigsten ursprünglichen Ziele visueller Programmierung ist die Vereinfachung des Programmiervorgangs. Durch die höhere Verständlichkeit und Anschaulichkeit visueller Programme soll es zudem möglich sein, schneller und mit weniger Einarbeitungszeit Programme zu erstellen und dabei Fehler zu vermeiden.

Aus diesen Zielen leiten sich verschiedene Zielgruppen ab, für die visuelle Programmierung im Besonderen gedacht ist:

- Kinder, die mit Hilfe visueller Programmierung die Verwendung von Computern und das Programmieren auf spielerische Weise lernen sollen
- Programmiereinsteiger, die durch die grafische Darstellung von Programmen Strukturen und Konzepte des Programmierens lernen sollen
- Endbenutzer, die auf einfache Weise Programme an ihre Bedürfnisse anpassen oder selbst kleinere Programme erstellen können sollen, um ihre Aufgaben zu erledigen
- Spezialisten auf bestimmten Fachgebieten, die keine zusätzlichen Programmierfähigkeiten brauchen sollen, um Simulations-, Steuerungs- und Messprogramme für ihr Fachgebiet zu erstellen
- Programmierer, die mit neuen Ansätzen schneller und fehlerfreier Programme erstellen können sollen

*E-Mail: philipp.merkel@uni-ulm.de

## 3 Typen von Visual Programming Languages

Im Laufe der Zeit haben sich verschiedene Grundformen von Visual Programming Languages etabliert, in die sich die meisten dieser Sprachen einordnen lassen. Im Folgenden werden die wichtigsten vorgestellt.

### 3.1 Flowchart-basierte VPLs

Eine der ältesten Klassen visueller Programmiersprachen stellt die der Flowchart-basierten VPLs dar. Ziel ist es hierbei, klassische Flowcharts oder vergleichbare Diagramme, etwa Struktogramme, ausführbar zu machen. Mit Flowcharts lässt sich der Kontrollfluss des Programms anschaulich spezifizieren, die Ausführungslogik ist vergleichbar mit klassischen imperativen Programmiersprachen. Die Speicherung von Daten erfolgt in Variablen, die im Programmverlauf gesetzt und ausgelesen werden können, was im Flowchart durch entsprechende (textuelle) Anweisungen dargestellt wird.

Aufgrund der schlechten Handhabbarkeit von Flowcharts und der einfacheren Verwendbarkeit textueller, imperativer Programmiersprachen sind solche VPLs jedoch heute kaum noch in Gebrauch.

### 3.2 Regelbasierte VPLs

Eine weitere wichtige Gruppe von VPLs ist die der regelbasierten Programmiersprachen. In diesen werden bestimmte Regeln angegeben, die Vorher-/Nachher-Bedingungen beinhalten. Wann immer im Verlauf der Ausführung des Programms die Situation auftritt, dass die Vorher-Bedingung gilt, wird vom System automatisch dafür gesorgt, dass der in der Nachher-Bedingung angegebene Zustand eintritt. In manchen Sprachen wird anstatt einer Nachher-Bedingung oder auch zusätzlich dazu eine Folge von Befehlen oder Aktionen angegeben, die ausgeführt werden sollen, sobald die Vorher-Bedingung gilt.

Diese Art der Programmierung wird häufig bei Sprachen eingesetzt, bei denen die Haupttätigkeit der damit erzeugten Programme darin besteht, Objekte auf einer Art Spielfeld zu bewegen. Oft sind solche Sprachen für Kinder gedacht, die damit kleine Spiele und Simulationen erstellen können. Die Bedingungen werden dabei meist in Form von Bildern angegeben, die den Zustand eines Ausschnitts des Spielfelds vor bzw. nach der Anwendung der Regel beschreiben, sog. Graphical Rewrite Rules (vgl. [Smith et al. 1994]). Beispiele für solche Sprachen sind AgentSheets (vgl. [Repenning and Citrin 1993]) und StageCast Creator (ehemals KidSim/Cocoa).

### 3.3 VPLs mit Programming by Example

Programming by Example ist ein Programmierparadigma, das bei visuellen Programmiersprachen häufig in Verbindung mit anderen Konzepten, etwa regelbasierter Programmierung, anzutreffen ist. Für Programming by Example wird die Entwicklungsumgebung in einen Aufnahmemodus geschaltet, in dem der Programmierer dann an einem bestimmten Objekt eine Aktion durchführt. Die Programmierumgebung wertet nun die durchgeführte Aktion aus, wobei sie automatisch abstrahiert und manchmal auch parametrisiert wird. Hierbei wird ein Programmstück erzeugt, das dann auch mit anderen Objekten bzw. veränderten Parametern erneut ausgeführt werden kann. (vgl. [Cypher et al. 1993])

Die Art des erzeugten Codes hängt stark von der Programmiersprache ab. Häufig werden Vorher-Nachher-Regeln erstellt, die dann vom Programmierer weiter bearbeitet werden können. In anderen Fällen werden textuelle Codefragmente erzeugt, was jedoch keine reine visuelle Programmierung darstellt, da zwar die Erstellung des Codes visuell (durch das Beispiel) erfolgt, der Code dann jedoch in Textform weiterbearbeitet werden muss.

Die regelbasierte VPL KidSim, die zuvor vorgestellt wurde, ist ein Beispiel für eine Sprache, bei der auch Programming by Example verwendet werden kann, um die Regeln zu erstellen (vgl. [Smith et al. 1994]).

### 3.4 Tabellen-/Zellenorientierte VPLs

Eine weitere Klasse der visuellen Programmiersprachen stellen die tabellen- bzw. zellenorientierten VPLs dar. Diese Sprachen bedienen sich dem klassischen Konzept einer Tabellenkalkulation. Dem Programmierer stehen Zellen zur Verfügung, die sowohl zur Eingabe, Ausgabe und Zwischenspeicherung von Daten dienen als auch die Programmlogik selbst enthalten. Die Zellen sind entweder als Tabelle angeordnet oder frei positionierbar und können häufig neben Text auch Grafik und Steuerelemente aufnehmen.

Die Programmierung in einer solchen Sprache erfolgt funktional durch Festlegen von Formeln für einzelne Zellen. Diese können andere Zellen referenzieren und produzieren ein Ergebnis, das als Wert der die Formel enthaltenden Zelle dient. Dies kann, wenn die Sprache Grafik vorsieht, auch ein grafisches Ergebnis sein.

Obgleich hauptsächlich mit Text gearbeitet wird, sind solche Sprachen als VPLs zu betrachten, da die Einordnung des Texts in die verschiedenen Zellen und somit die Anordnung auf der Fläche für die Programmsemantik von Bedeutung ist, auch wenn manche solcher Sprachen Grenzfälle in der Einordnung darstellen.

Ein Vertreter dieser Klasse von Sprachen ist Forms/3 (vgl. [Burnett 1999]), aber auch viele klassische Spreadsheet-Systeme wie Microsoft Excel lassen sich in diese Kategorie einordnen, wenngleich sie einen wesentlich eingeschränkteren Funktionsumfang bieten.

### 3.5 Datenflussorientierte VPLs

Die heute gebräuchlichste Art der visuellen Programmierung ist die Verwendung von datenflussorientierten Sprachen. Bei der Datenflussprogrammierung ist das Programm als gerichteter Graph zu betrachten, dessen Knoten als Prozessoren dienen, die Daten aufnehmen, verarbeiten und ausgeben. Die Kanten stellen Leitungen dar, über die die Daten von einem Prozessor zum anderen gelangen. Die Semantik eines solchen Programms ist folgendermaßen: Sobald an allen Eingangskanten eines Knotens Daten anliegen, werden diese vom Knoten verarbeitet. Dies geschieht, indem das System - mit den Eingangsdaten als Argumenten - eine für den Knoten spezifische Funktion berechnet. Das Ergebnis der Funktion kann aus mehreren Werten bestehen, die nun an den Ausgangskanten anliegen und von den verbundenen Knoten weiterverarbeitet werden. Die Daten gelangen über spezielle Eingabeknoten in das Programm und werden am Ende der Verarbeitung über spezielle Ausgabeknoten ausgegeben. Die Programmlogik besteht also aus einem Fluss von Daten aus den Eingabe- in die Ausgabeknoten (vgl. [Johnston et al. 2004]).

Die datenflussorientierte Programmierung ist der funktionalen Programmierung sehr ähnlich. Durch die Darstellung der Programme als Graphen ist es jedoch wesentlich einfacher, Funktionen zu verwenden, die mehrere Rückgabewerte liefern.

Zwischen den einzelnen Verarbeitungsschritten speichern die Knoten, wie bei funktionalen Sprachen auch üblich, keine Daten und sind somit zustandslos. Damit sind datenflussorientierte Sprachen für verteilte Systeme und Mehrprozessorsysteme geeignet, da theoretisch jeder einzelne Knoten in einem separaten Prozess oder auf einem separaten Rechner laufen kann und kein übergeordneter Zustand des gesamten Systems notwendig ist.

Es existieren zahlreiche, auch kommerziell erfolgreiche, datenflussorientierte VPLs, bekannte Vertreter sind LabView von National Instruments, Quartz Composer von Apple (vgl. [Apple Inc. 2007]), Max/MSP und Pure Data.

### 3.6 Hybride Sprachen

Viele Visual-Programming-Systeme können nicht nur durch eine der oben genannten Klassen charakterisiert werden, sondern kombinieren mehrere Formen und werden als hybride Sprachen bezeichnet. So verwenden z.B. viele Programming-By-Example-Systeme regelbasierte Sprachen zur Darstellung und Weiterverar-

beitung des erzeugten Codes. Auch gibt es Fälle, in denen textuelle und visuelle Sprachen kombiniert werden, indem z.B. Klassen eines objektorientierten Systems oder einzelne Funktionen einer funktionalen Programmiersprache wahlweise verbal oder visuell programmiert und verschiedenartige Komponenten zu einem Gesamtprogramm zusammengesetzt werden können. Diese Form der hybriden Sprachen ist derzeit jedoch (noch) nicht von praktischer Relevanz.

## 4 Gemeinsame Eigenschaften

Trotz der zahlreichen verschiedenen Formen, in denen VPLs auftreten, gibt es einige Gemeinsamkeiten, die fast alle diese Sprachen bzw. ihre VPEs verbinden.

Bei vielen VPLs ist die Programmierung sehr User-Interface-zentriert und eine Trennung von Benutzeroberfläche und Programmlogik nur schwer möglich. So hat z.B. bei der Datenfluss-Programmiersprache LabView jede Programmkomponente ein „Front Panel“ (vgl. [Kalkman 1995]), das zum einen ihre Benutzeroberfläche darstellt, zum anderen eine Programmierschnittstelle für andere Programmteile ist. Bei der Kinder-Simulationssprache KidSim haben alle Objekte des Programms eine visuelle Repräsentation auf dem „Spielfeld“, mit der sie auch im Code dargestellt werden.

Eine wichtige Fähigkeit fast aller VPEs visueller Sprachen ist zudem, dass die Programme in der Entwicklungsumgebung selbst ausgeführt werden oder werden können. Oft ist es auch möglich, den jeweiligen Programmzustand „live“ im visuellen Programmcode zu verfolgen oder sogar den Programmcode während der Ausführung zu verändern. In einigen VP-Systeme läuft das Programm sogar immer, wenn es in der Entwicklungsumgebung geladen ist und Programmieren und Testen erfolgt gleichzeitig.

## 5 Vor- und Nachteile

Ein großer Vorteil der visuellen Programmierung gegenüber der Programmierung mit klassischen verbalen Sprachen ist der geringere Lernaufwand. Anstatt sich die Namen von zahlreichen verbalen Befehlen auswendig merken und Buchstabe für Buchstabe korrekt eingeben zu müssen, kann man bei einer VPL oft aus einer Menge von grafischen Symbolen oder Textbausteinen wählen, die für die verschiedenen Programmierkonstrukte stehen. Bei Programming by Example geht dies sogar so weit, dass man sich überhaupt keine Sprachkonstrukte mehr merken muss, da diese von der Entwicklungsumgebung automatisch erstellt werden.

Da die einzelnen Programmierkonstrukte bei VPLs nicht selbst eingetippt, sondern aus Katalogen ausgewählt werden, treten praktisch keine Syntaxfehler mehr auf. Auch Fehler, die durch falsche Kombination von Befehlen entstehen, werden vermieden, da bereits bei der Erstellung des Programms die Programmierumgebung das fehlerhafte Zusammensetzen von Objekten verhindern kann.

Allerdings kann eine reine grafische Darstellung von Befehlen auch negative Folgen haben, wenn komplexe Funktionen durch Symbole dargestellt werden sollen. Hierbei ist es oft nicht möglich, selbsterklärende Symbole zu erstellen, und eine textuelle Repräsentation der Funktion wäre leichter zu merken und zu verwenden. Hier zeigt sich, dass sich der zwanghafte Verzicht auf Text negativ auf die Verwendbarkeit einer Programmiersprache auswirken kann. So sind die einzelnen Funktionen in der VPL von Quartz Composer, die als Text angegeben sind, verständlicher als manches kryptische Symbol aus LabView (vgl. Abb. 1).

Grundsätzlich unterstützt jedoch die grafische Darstellung den Programmierer dabei, ein Verständnis vom Konzept der Programmiersprache zu bekommen. So lassen sich viele Konzepte durch Grafik leichter repräsentieren als durch Text, etwa ist ein Kästchen, aus dem ein Pfeil auf ein anderes Kästchen zeigt eine verständlichere Darstellung eines Pointers als `int* a = &b;`. Diese Anschaulichkeit macht in visuellen Sprachen beschriebene Programme besonders verständlich und in gewissem Grad auch

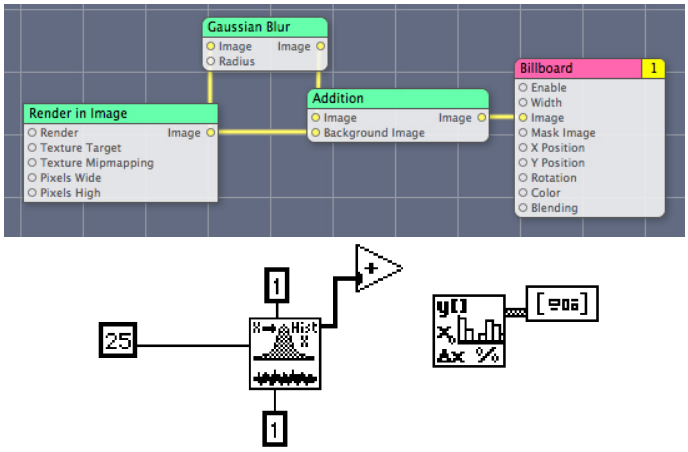


**Abbildung 1:** *Vergleich Quartz Composer/LabView (Quelle: [Apple Inc. 2007, „Creating a Glow Filter“], [Schiffer 1996, Abb. 1])*

selbsterklärend.

Allerdings werden durch die grafische Darstellung des Codes stets nur bestimmte Programmeigenschaften besonders gut und anschaulich repräsentiert. Welche dies sind, hängt von der Art des Programmiersystems ab: So wird bei Datenflusssprachen etwa der Verlauf der Daten durch die einzelnen funktionalen Prozessoren wesentlich intuitiver dargestellt, als dies bei einer funktionalen Schreibweise des Programms geschehen würde. Bei Flowchart-Sprachen wird der Kontrollfluss klar dargestellt, insbesondere bei Sprüngen, da anstatt eines Verweises auf die Stelle, zu der gesprungen werden soll, einfach ein Pfeil an diese Stelle gezeichnet wird. Für andere, durch die grafische Darstellung nicht visualisierte Eigenschaften sind jedoch weiterhin Annotationsmöglichkeiten notwendig, die Kommentaren in klassischen Programmiersprachen entsprechen. Dass manche visuelle Programmiersprachen solche Möglichkeiten nicht bieten, ist ein gängiger Kritikpunkt und bei der Entwicklung aufwändigerer Projekte hinderlich.

Immer wieder als Vorteil visueller Programmierung genannt wird die Möglichkeit des explorativen Programmierens. So kann man als Programmierer mit dem Programmieren beginnen, bevor man sich eine genaue Lösung des Problems überlegt hat und sein Programm Schritt für Schritt aufbauen und dabei testen, wie es funktioniert. Dies kann jedoch auch als Nachteil gesehen werden: Während es bei kleinen Projekten oftmals praktisch sein kann, einfach darauf los zu programmieren, ohne sich vorher viele Gedanken über die Struktur seines Programms machen zu müssen, kann dies bei größeren Projekten schnell zu einem unstrukturierten, schlecht wart- und erweiterbaren Code führen.

Das zeigt bereits ein weiteres Problem der visuellen Programmierung auf, die schlechte Skalierbarkeit. Während sich kleine Programme mit VPLs häufig schnell und einfach erstellen lassen, sind große Projekte oft schwer zu realisieren, da der Code schnell zu einem unübersichtlichen Wirrwarr aus Linien und Kästchen werden kann.

Hier müssen visuelle Programmiersprachen Mittel anbieten, mit denen man Teile des Codes zu Einheiten zusammenfassen kann, sodass ein modularer Aufbau des Projekts möglich ist und die einzelnen Bestandteile nur aus wenigen Codekomponenten bestehen. So könnte z.B. in einer Datenflusssprache ein Teilgraph zu einem einzigen Knoten gekapselt werden, der die Eingänge und Ausgänge des Teilgraphen besitzt und von außen wie ein normaler Funktionsknoten wirkt, im Inneren jedoch aus den Knoten und Verbindungen des Teilgraphen besteht (vgl. [Burnett 1999]). In anderen VPL-Typen ist ähnliches möglich und wird häufig auch eingesetzt.

Die Unübersichtlichkeit größerer visuell beschriebener Programme resultiert oft auch aus ihrer „unökonomischen Bildschirmnutzung“ (vgl. [Schiffer 1996]). So nimmt ein Programm in einer VPL meist eine wesentlich größere Fläche ein als ein textueller Programmcode gleicher Semantik. Somit ist es schon bei nur geringfügig komplexen Programmen schwierig, den Überblick über die Struktur des Codes zu behalten, da stets nur ein kleinerer Teil des Programms sichtbar ist, oder die Darstellung stark verkleinert werden muss. Dies ist insbesondere bei Sprachen, die verzweigte Strukturen mit viel Leerraum beinhalten - etwa bei Datenfluss- oder Flowchart-Sprachen - der Fall, weniger bei regel- oder zellenbasierten Sprachen, die sich recht kompakt darstellen lassen. Allerdings beginnt dieses Problem mit den sinkenden Preisen für große, hochauflösende Bildschirme und dem vermehrten Einsatz von Mehrbildschirmumgebungen an Wichtigkeit zu verlieren. Außerdem lässt sich der Platzbedarf der Programme durch die oben bereits angesprochenen Maßnahmen der Modularisierung verringern, was in vielen Fällen sogar zu einem übersichtlicheren, kompakteren und strukturierteren Programm führen kann als bei einer Realisierung mit einer textuellen Sprache, bei der eine solche Modularisierung aufgrund des nicht vorhandenen Größenproblems möglicherweise als nicht notwendig erachtet würde.

Ein weiteres Problem der grafischen Programmierung ist, dass die in VPLs geschriebenen Programme häufig eine mehrdimensionale, nichtlineare Struktur aufweisen, in vielen Anwendungsgebieten jedoch ein linearer Kontrollfluss für eine deterministische Semantik notwendig ist. Das Mapping der mehrdimensionalen Programmstruktur in solch einen linearen Kontrollfluss ist insbesondere bei Datenflusssprachen häufig schwierig und mit zusätzlichem Aufwand für den Programmierer verbunden. So muss z.B. bei Quartz Composer, mit dem unter anderem grafische Objekte auf eine Zeichenfläche gezeichnet werden können, unabhängig vom Fluss der Daten, der ja theoretisch parallel erfolgen kann, eine Reihenfolge festgelegt werden, in der die einzelnen Objekte gezeichnet werden sollen. Hierzu müssen manuelle Ordnungszahlen für die einzelnen zeichnenden Funktionsblöcke festgelegt werden, bei textuellen Sprachen würde sich dies automatisch aus der Reihenfolge der Befehle im Code ergeben.

## 6 Resultierende Anwendungsgebiete

Die angesprochenen Vor- und Nachteile zeigen, dass VP zwar zahlreiche positive Eigenschaften hat, jedoch vor allem bei großen und komplexen Projekten Probleme auftreten können. Dies hat Folgen für den heutigen Einsatz visueller Programmiersprachen.

So gibt es heutzutage so gut wie keine Allround-VPLs mehr, die versuchen, textuelle Sprachen komplett zu verdrängen und für alle Arten von Programmierproblemen einsetzbar zu sein. Stattdessen gibt es anwendungsspezifische Programmiersprachen, die für bestimmte Einsatzgebiete und Benutzergruppen gedacht sind. Dabei werden Funktionen, die in den Einsatzgebieten nicht benötigt werden, nicht bereitgestellt und Funktionen, die man in klassischen Programmiersprachen selbst entwickeln müsste, werden als fertige Funktionsbausteine in die Sprache integriert oder bilden gar ihre Grundlage und vermindern so den Umfang und die Komplexität der vom Benutzer erstellten Programme. Auch kann sich die Darstellung des Programms nach der Zielgruppe richten und spezielle, den Benutzern bereits bekannte Symbole verwenden.

Ein Einsatzgebiet solcher anwendungsspezifischer Sprachen ist der Schulunterricht. Kinder können mit speziellen regelbasierten Sprachen oder durch Programming-By-Example kleine Simulationen zu erstellen, um so natur- oder gesellschaftswissenschaftliche Phänomene durch eigenes Experimentieren zu verstehen.

Auch in professionellerem Umfeld wird visuelle Programmierung eingesetzt: spezielle datenflussbasierte Programmiersysteme wie Pure Data oder Quartz Composer, die auf die Verarbeitung, Synthetisierung und Transformation von Multimediadaten spezialisiert sind, dienen Audio-, Video- und Multimediakünstlern zur Erstellung digitaler Kompositionen. Im technischen Bereich wird häufig die kommerzielle Datenflusssprache LabView eingesetzt, um Daten zu erfassen und zu verarbeiten.

Grundsätzlich werden VPLs heute - abgesehen von der Forschung - also hauptsächlich in Bereichen eingesetzt, in denen Nicht-Programmierer ausführbare Programme erstellen. Dies muss nicht unbedingt End User Programming sein, da die Benutzer der visuellen Programmiersysteme auch Experten auf anderen Gebieten sein können, die Programme für Nicht-Experten schreiben, oder Künstler, die interaktive Kunst für die eigentlichen Endbenutzer, die Betrachter, schaffen möchten.

## 7 Schlussfolgerung

Die Vorteile visueller Programmierung zeigen sich also vor allem dann, wenn man Sprachen verwendet, die für die jeweiligen Einsatzbereiche spezialisiert sind, oder die VP nur für kleinere Teilbereiche eines größeren Projekts, etwa das User Interface, einsetzt. Allround-VPLs haben derzeit und wohl auch in der näheren Zukunft keine praktische Relevanz.

Sprachparadigmen, mit denen sich auch in einer verbalen Sprache schnell und einfach Programme schreiben lassen, wie etwa die imperative Programmierung, gewinnen durch eine Umsetzung auf visueller Ebene meist wenig, oder werden durch den größeren Platzbedarf und das Problem, die Schlüsselwörter in Symbole umsetzen zu müssen, gar noch komplizierter und umständlicher zu verwenden als vergleichbare textuelle Sprachen - zumindest was größere Projekte betrifft. Andere Konzepte, bei denen eine textuelle Umsetzung schwierig ist, wie etwa Datenflusssprachen, bergen jedoch ein großes Potential, in speziellen Anwendungsgebieten eine einfache, fehlervermeidende und schnelle Entwicklung von Programmen, auch durch Endbenutzer oder andere Nicht-Programmierer, zu ermöglichen.

## Literatur


APPLE INC. 2007. Quartz composer user guide. http://developer.apple.com/documentation/GraphicsImaging/Conceptual/QuartzComposerUserGuide/.

BURNETT, M. 1999. Visual programming. In *Encyclopedia of Electrical and Electronics Engineering*, J. G. Webster, Ed. John Wiley & Sons Inc., New York.

CYPHER, A., HALBERT, D. C., KURLANDER, D., LIEBERMAN, H., MAULSBY, D., MYERS, B. A., AND TURRANSKY, A. 1993. *Watch What I Do: Programming by Demonstration*. The MIT Press, http://acypher.com/wwid/, ch. Introduction.

JOHNSTON, W. M., HANNA, J. R. P., AND MILLAR, R. J. 2004. Advances in dataflow programming languages. *ACM Comput. Surv. 36*, 1, 1–34.

KALKMAN, C. J. 1995. Labview: A software system for data acquisition, data analysis, and instrument control. *Journal of Clinical Monitoring and Computing 11*, 1 (Januar), 51–58.

REPENNING, A., AND CITRIN, W. 1993. Agentsheets: Applying grid-based spatial reasoning to human-computer interaction. In *1993 IEEE Workshop on Visual Languages, Bergen, Norway*, IEEE Computer Society Press, 77–82.

SCHIFFER, S. 1996. Visuelle Programmierung - Potential und Grenzen. In *GI Jahrestagung 1996*, 267–286.

SMITH, D. C., CYPHER, A., AND SPOHRER, J. 1994. KidSim: programming agents without a programming language. *Commun. ACM 37*, 7, 54–67.