Inhalt – Vorlesungsteil Visuelle Programmierung

1. Übersicht 2
1.1. Einsatzgebiete visueller Sprachen 2
1.2. Eigenschaften visueller Sprachen 2
1.3. Abgrenzung visueller Sprachen 3
1.4. Ursprünge visueller Programmierung 4
1.5. Logo und Schildkrötengrafik 4
1.6. Scratch 5
1.7. EToys 5
2. Verschiedene Paradigmen der visuellen Programmierung 7
2.1. Steuerflussorientierte Systeme 7
2.2. Objektorientierte Systeme 8
2.3. Datenflussorientierte Systeme 8
2.4. Regelorientierte Systeme 10
2.5. Constraintsorientierte Systeme 11
2.6. Formularorientierte Systeme 12
2.7. Beispielorientierte Systeme 12
3. Fazit 12

1. Übersicht

1.1. Einsatzgebiete visueller Sprachen

Selbst einfache Programmiersprachen stellen Programmieranfänger und Endanwender häufig vor schwierige Probleme. Der Einstieg in die textuelle Programmierung wird unter anderem durch folgende Faktoren erschwert. Zum einen muss man die exakte Syntax der Sprache und der einzelnen Anweisungen kennen. Wenn Schlüsselkonzepte, wie z.B. die for-Schleife, nicht bekannt sind, dann kann auf diese natürlich nicht zurückgegriffen werden. Zudem muss die genaue Anordnung und Semantik von Ablaufstrukturen, Anweisungen und Funktionen bekannt sein. Sicherlich haben Sie auch schon einmal in einer C- oder Java-for-Schleife die Reihenfolge von Schleifenbedingung und Schleifeninkrement vertauscht. Natürlich schleichen sich auch schnell Tippfehler ein – vergessene Kommas oder Semikolons haben schon für manche Verzweiflung gesorgt! Auch Klassen-, Objekt- und Variablennamen muss man sich genau merken und darf sie nicht durcheinander bringen. Bei der Programmierung visueller Anwendungen (Benutzungsoberfläche, Grafikprogramme, Multimediaanwendungen, Animation usw.) ist die textuelle Eingabe von Werten häufig eine schwer interpretierbare Abstraktion – so kann man sich unter `preis = 1.99` sicherlich besser vorstellen was gemeint ist als unter `kastenFarbe = new Color (22 , 12, 214)`. Gleiches gilt für grafische Attribute wie Position, Größe, Rotation, Texturen usw.

Einige dieser Probleme versucht man mit Hilfe visueller Programmierung in den Griff zu bekommen. Dabei zielt die visuelle Programmierung vor allem auf Anfänger, Endanwender und die Entwicklung graphisch orientierter Anwendungen (z.B. Multimediasysteme, E-Learning, 3D-Welten, Bildbearbeitung, Spiele, einfache Apps und Webanwendungen). So erhofft man sich von visuellen Programmiersprachen einen niedrigschwelligen Einstieg in die Programmierung. Ergebnisse werden schneller sichtbar und bereits bei der Entwicklung (be-)greifbar. Das Anordnen visueller Elemente fördert zudem eine spielerische Herangehensweise und eignet sich gut zum Experimentieren, insbesondere weil die Entwicklungsumgebungen unzulässige Anordnungen und Konfigurationen von visuellen Elemente gar nicht erst zulassen. So lassen sich insbesondere lästige Syntaxfehler vermeiden.

Wir werden sehen, dass es viele unterschiedliche Paradigmen der visuellen Programmierung gibt. Dabei wird auch deutlich, dass der imperative Programmierstil, wie man ihn etwa in prozeduralen und objektorientierten Sprachen findet, nicht die einzige Herangehensweise ist – aber dies ist Ihnen aus der Logikprogrammierung und der funktionalen Programmierung bereits bekannt.

Während die ersten Ansätze der visuellen Programmierung bereits in die 1960er Jahre zurückgehen (SketchPad von Ivan Sutherland, GRaphical Input Language GRAIL von Alan Kay, oder das von Seymour Papert mitentwickelte Logo), erleben visuelle Sprachen derzeit eine Renaissance im Umfeld der plattformübergreifenden Appentwicklung, insbesondre bei der Spieleentwicklung, sowie der Festlegung von Algorithmen mithilfe visueller Codebausteine (Zusammensetzen von Programmen wie mit Lego-Steinen). In diesem Zusammenhang spielt auch die frühe Programmierausbildung in der Schule eine Rolle, teils bereits in der Grundschule. Die „Hour of Code“ (http://hourofcode.com/) Bewegung vermittelt den leichten Einstieg in die Programmierung und erreicht bereits viele Millionen Schüler. Viele der benutzen Programmiersprachen sind teils visuell orientiert.

1.2. Eigenschaften visueller Sprachen

Der offensichtlichste Unterschied visueller Sprachen im Vergleich zu „klassischen“ Programmiersprachen ist der visuelle Aspekt. Andererseits könnte man aber auch argumentieren, dass

alle Programmiersprachen visuell sind, denn auch die textliche Darstellung ist letztlich etwas visuelles, insbesondere wenn dann noch Syntaxhighlighting und pragmatische Aspekte der Programmstrukturierung (Einrückungen, Leerzeilen) dazukommen. Doch diese visuellen Auszeichnungen haben keinen Einfluss auf die Syntax und Semantik des Programms: Compiler und Interpreter ignorieren Leerzeichen, Leerzeilen, Schrifteinstellungen usw.

Mit visueller Programmierung ist aber gemeint, dass die visuellen Zeichen auch tatsächlich den Algorithmus bzw. das Systemverhalten (zumindest in Teilen) festlegen. Der Grad der visuellen Ausdruckskraft hängt davon ab, inwieweit die folgenden Elemente eine Rolle spielen:

- Graphische Komponenten: Diagramme, Piktogramme, Farben
- Geometrische Eigenschaften: Form, Größe, Seitenverhältnisse
- Topographische Eigenschaften: Verbindungen, Überschneidungen, Kontaktpunkte
- Typographische Eigenschaften: Seitengestaltung, Schriftstil, Einrückungen

Allgemein können wir sagen, dass eine visuelle Sprache eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung ist. Auf Programmiersprachen bezogen, also auf notationelle Systeme zur Beschreibung von Berechnungen in durch Maschinen und Menschen lesbarer Form, können wir zur folgenden **Definition** kommen:

> *Visuelle Programmierung bedeutet, dass die Beschreibung der Rechenschritte durch visuelle Zeichen erfolgt. Die Beschreibung von Programmen und Systemverhalten geschieht primär durch Verwendung visueller Komponenten. Einzelne graphische Elemente sowie die korrekte Verknüpfung und Konfiguration bilden die Syntax. Die graphischen (Farben, Icons), geometrischen (Form, Größe, Seitenverhältnisse) und topologischen (Verbindungen, Überlagerungen, Berührungen) Eigenschaften legen die Semantik fest.*

Visuelle Zeichen können statisch oder dynamisch sein. Statisch bedeutet, dass ihre Repräsentation dauerhaft ist – ein Kreis behält z.B. seine Form, Größe und Position bei. Dynamisch bedeutet, dass die bei der Programmierung eingesetzten Zeichen flüchtig oder nicht dauerhaft in gleicher Form bleiben, z.B. das Programmieren von Bewegungsabläufen durch Verschieben von Objekten oder das Aufzeichnen von Bewegungsabläufen mit der Maus. Statisch ist z.B. auch das geschriebene Wort „Hallo“, während das ausgesprochene „Hallo“ dynamisch ist. Das geschriebene „Hallo“ können Sie leicht in „Hello“ umwandeln, um das dynamisch aufgezeichnete „Hallo“ in ein „Hello“ umzuwandeln müssten Sie eine neue Audioaufzeichnung starten.

1.3. Abgrenzung visueller Sprachen

Viele Programmierumgebungen (Integrated Development Environments, IDEs) werben damit, visuelle Programmierung zu ermöglichen (Visual Studio, Eclipse), da mit ihnen graphische Benutzungsoberflächen ohne Programmierung erstellt werden können. Jedoch wird dabei wiederum textueller Programmcode erzeugt und die eigentliche Ablauflogik wird ebenfalls textuell festgelegt. Im strengen Sinne handelt es sich also nicht um visuelle Programmierung. Im weiteren Sinne muss aber anerkannt werden, dass zumindest das (textuelle) Programm mit visuellen Mitteln erstellt wird und dass die Anordnung von visuellen GUI-Objekten wie Buttons, Schiebereglern, Listen usw. häufig natürlicher ist wenn ein graphischer Editor verwendet wird. Die Generierung von Code mit Hilfe graphischer Eingabehilfen (z.B. Codevervollständigung, Drag&Drop Codebausteine) sowie pragmatische Auszeichnungen (verschiedene Farben für Schlüsselwörter, Variablen, Konstanten, Kommentare usw.) gelten jedoch nicht als visuelle Programmierung, da sie keinen Einfluss auf den Programmablauf haben sondern vor allem Hilfestellungen für den Entwickler darstellen.

1.4. Ursprünge visueller Programmierung

Bereits 1963 stellte Ivan Sutherland einen interaktiven Grafikeditor (Sketchpad) mit objektorientierten Ansatz vor. Mit einem Lichtstift konnten Formen auf einem Monitor gezeichnet werden – vom Bedienkonzept ein früher Vorläufer heutiger Tabletcomputer. Der objektorientierte Ansatz erlaubte das Festlegen von Vorlagen und parametrisierten Instanzen geometrischer Formen. Das Festlegen von Abhängigkeiten zwischen einzelnen Elementen ist eine erste Form constraints-orientierter visueller Programmierung.

Das Programmieren mit Flussdiagrammen, die Abläufe und Zusammenhänge visuell festlegen, wurde bereits in 1968 von Alan Kay konzeptionell mit der GRaphical Input Language (GRaiL) vorgestellt. Hier finden Sie ein Video, das den Ansatz zeigt:

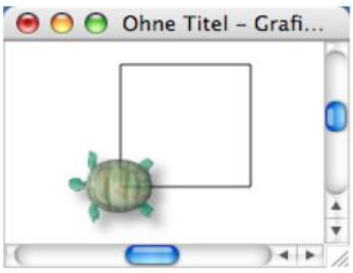
https://www.youtube.com/watch?feature=player_embedded&v=QQhVQ1UG6aM

1.5. Logo und Schildkrötengrafik

1967 entwickelten Daniel G. Bobrow, Wally Feurzeig, Seymour Papert und Cynthia Solomon die Programmiersprache Logo. Diese Sprache sollte speziell beim Programmierlernen helfen, denn sie versprach zum einen den niedrigschwelligen Einstieg und zum anderen begrenzte sie nicht die Tiefe der Möglichkeiten („no treshold, no ceiling“). Semor Papert verband mit Logo auch den von ihm entwickelten lerntheoretischen Ansatz des Konstruktionismus. Dieser geht davon aus, dass Lernen durch aktives Handeln erfolgt und Wissen immer eigenständig aufgebaut wird. Wissen wird also nicht einfach transferiert (wie beim „Nürnberger Trichter“) sondern von jedem Individuum eigenstädnig re-konstruiert. Der Konstruktionismus gehört damit in die Denkschule des Konstruktivismus (wir konstruieren unser eigenes Wissen und eigenen Sichtweisen).

Ein wichtiges Konzept beim entdeckenden Lernen ist dabei die „Schildkröten“-Grafik bzw. „Turtle“-Grafik. Durch einfache Anweisungen lässt sich eine graphische Schildkröte über den Bildschirm bewegen. Durch das Zeichnen einer Spur können so komplexe Grafiken erzeugt und erlebt werden.

```
Forward 80    bzw.     fd 80
Right 90               rt 90
Forward 80             fd 80
Right 90               rt 90
Forward 80             fd 80
```



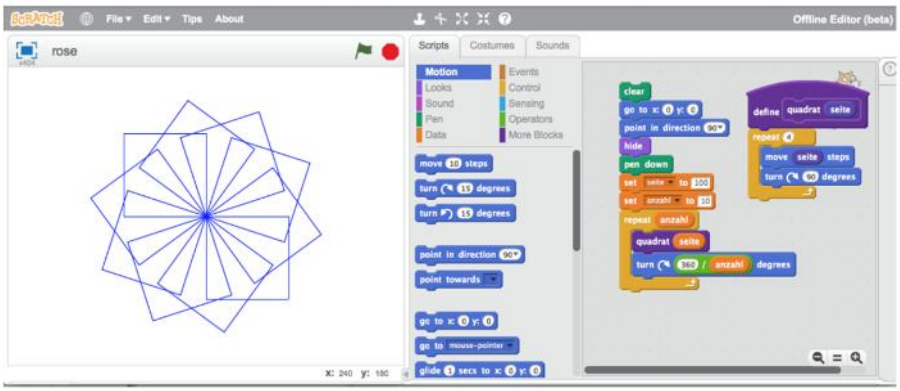
Spätere Logo-Implementierungen erlaubten das parallele Ausführen vieler autonomer Schildkröten (jetzt Agenten genannt), die auf ihre Umgebung reagieren konnten. Somit war die Simulation komplexer dynamischer Systeme, z.B. dem Schwarmverhalten von Vögeln, möglich. Das bereits kennengelernte Actors-Konzept ist eng mit diesen Ansätzen verwandt, denn auch die Actors agieren voneinander unabhängig und tauschen sich nur über Nachrichten aus.

Der Ansatz der „Schildkröten“-Grafik ist im Kontext visueller Programmierung vor allem deshalb interessant, weil viele visuelle Programmiersprachen dieses Konzept wieder aufgreifen. Zwei prominente Beispiele mit didaktischer Zielsetzung sind Scratch und E-Toys. Beide Sprachen unterstützen nach Tauber das entdeckende, spielerische Lernen, das Formulieren und Ausprobieren von Vermutungen, das Lernen aus Fehlern und deren Korrektur, die Zerlegung von Problemen in Teilprobleme und deren Lösung mit Hilfe von Prozeduren, ein Baukastenprinzip, d.h. die Erweiterung

der Sprache durch eigene problemspezifische Prozeduren, die Erarbeitung von programmiersprachlichen Konzepten mit der Schildkrötengrafik sowie einfaches interaktives Arbeiten mit sofortigem grafischen Feedback.

1.6. Scratch

Scratch ist eine visuelle Programmiersprache, die vor allem die Erstellung von Spielen und einfachen Multimediaanwendungen ermöglicht. Es gibt eine webbasierte Version, mit der man sofort los programmieren kann - http://scratch.mit.edu/



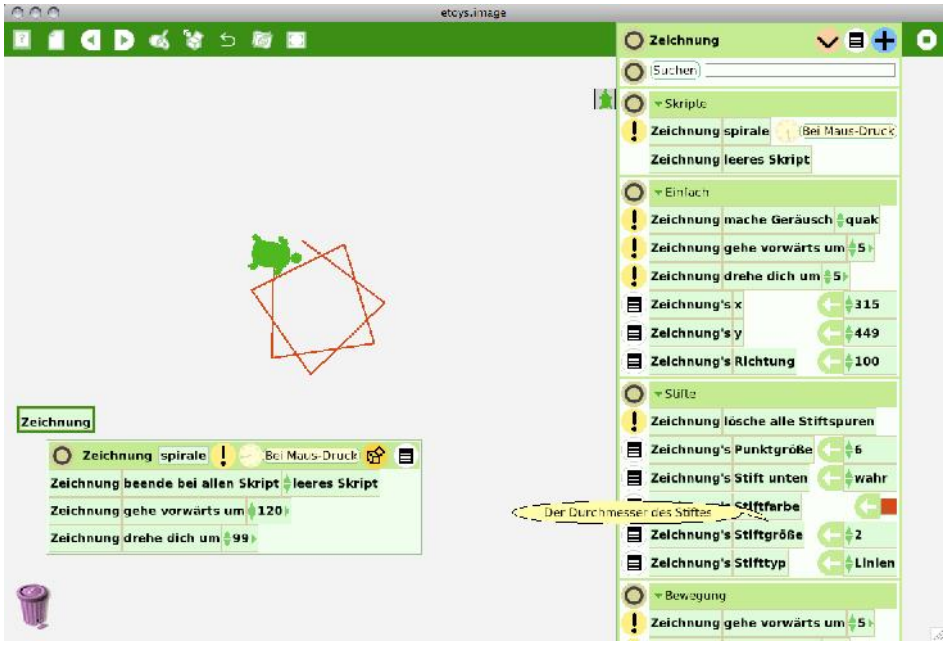
Auf einer Bühne fester Größe werden beliebig viele Sprites (eine Verallgemeinerung der Schildkröte) gesetzt und über zugeordnete Code-Module gesteuert. Die Code-Module werden durch visuelle Programmblöcke definiert. Zu den Anweisungen gehören unter anderem:

- Bewegung der Sprites (wie Gehe ... vorwärts, Drehe um, Zeige auf usw.)
- Aussehen (wie nächstes Kleid, ändere Grösse, verstecken, erscheinen usw.)
- Malstift (wie Stift runter, Stift hoch, wähle Stiftfarbe usw.)
- Steuerung (wie Wenn angeklickt, Warte, Wiederhole x mal, Fortlaufend wenn usw.)
- Fühler (Maus-Position, Farbe x wird berührt?, Entfernung von usw.)
- Zahlenoperationen (Addition, Subtraktion, Multiplikation, Division usw.)
- Nachrichtenversand (Benachrichtigungen lösen Events aus)

In Scratch können mehrere Objekte (Sprites) gleichzeitig angesprochen werden. Zwischen den Objekten können Nachrichten ausgetauscht werden. Jedem Objekt können eigene Code-Module zugeordnet werden, d.h. sie können autonom agieren und auf Ereignisse bzw. Nachrichten reagieren. Da die Objekteigenschaften und sogar der Programmcode während des Programmablaufs verändert werden kann, wird ein spielerisches und experimentelles Arbeiten geradezu herausgefordert.

1.7. EToys

Etoys setzt auf der Smalltalk-Umgebung Squeak auf und bietet somit einen konsequent objektorientierten Programmieransatz. Objekte können selbst gezeichnet und mit Variablen und Skripten zum Leben erweckt werden. Da sich Objekte mit einfachen Anweisungen über die Spielwiese bewegen lassen und dabei Zeichnungen anfertigen können, sind auch hier Schildkrötengrafiken möglich. Die Skripte werden ebenfalls visuell per Drag & Drop zusammengestellt, allerdings als reine Flussdiagramme, deren einzelne Anweisungs- und Ausdrucksblöcke allesamt gleich aussehen und somit die wirkungsvolle Pragmatik der Puzzle-Diagramme von Scratch nicht besitzen. Zudem werden syntaktisch unsinnige oder unvollständige Anordnungen nicht sofort erkenntlich.



Programmierung einer Spirale in EToys, hier umgesetzt mit Wiederholungsschleife.

In Etoys kann man durch eine einfache Anweisung festlegen, dass sich ein Objekt in Richtung eines anderen Objektes drehen soll. In Verknüpfung mit einer Test-Anweisung kann dies geschehen wenn sich ein Objekt zu weit von einem anderen entfernt hat, an die Grenzen der Spielwiese gerät oder auf eine bestimmte Farbe „tritt“. In einem weiteren, parallel ausgeführten Skript lässt sich dann festlegen wie sich das Objekt normal bewegen oder auf die Kollision mit anderen Objekten reagieren soll.

Durch die Interaktion mit der Spielwiese kann über die grafische Gestaltung der Umgebung das Verhalten der Objekte visuell definiert werden. Die Zeichnungen sind also mit Semantik belegt. Dies gilt auch für die Objekte selbst, die stets visuell repräsentiert sind. Bestimmte Farbpunkte der Objektzeichnung können bestimmen an welcher Stelle die Umgebung auf der Spielwiese analysiert wird. Eine Schildkröte in Etoys folgt also nicht nur festgelegten Kontrollstrukturen und Bewegungsmechanismen sondern analysiert auch die Umgebung und ist somit sehr gut für einfache Modellbildungen geeignet.

Dabei werden bereits einige Konzepte der Objektorientierung vermittelt, insbesondere der strukturelle Zusammenhang von Daten und Verhalten. Das Festlegen mehrerer Skripte, die parallel laufen und auch auf Ereignisse reagieren können, erlaubt zudem eine intuitivere Definition von Verhaltensregeln als dies bei (rein) imperativer Programmierung der Fall wäre. So formulieren Nicht-Programmierer die Regeln von Objekten eher natürlichsprachlich wie bei der Schildkröten-Grafik (z.B. „Objekt bewegt sich vorwärts“ oder „Objekt bewegt sich in Richtung X“) und nicht mathematisch (x:= x + 5).

2. Verschiedene Paradigmen der visuellen Programmierung

Mit Scratch und Etoys haben wir bereits verschiedene Paradigmen in Aktion gesehen. Während beide die Steuerung von Sprites über Ablaufblöcke ermöglichen und damit einem steuerflussorientierten Paradigma folgen, betont Etoys zusätzlich objektorientiere Ansätze, indem es das Anlegen von Eigenschaften und Methoden für einzelne Objekte auf der Spielwiese besonders einfach macht.

Es gibt eine Reihe weiterer Paradigmen visueller Programmieransätze (Schiffer, 1998; Ricardo Baeza-Yates, o.J.):

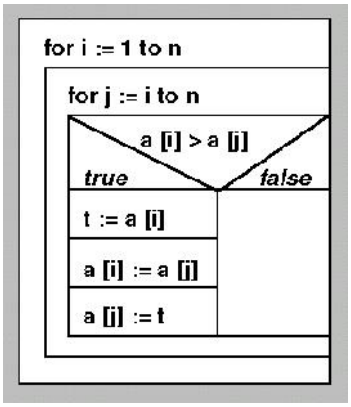
- Kontrollflussorientierte Systeme
- Objektorientierte Systeme
- Datenflussorientierte Systeme
- Funktionsorientierte Systeme
- Regelorientierte Systeme
- Constraint-orientierte Systeme
- Beispielorientierte Systeme
- Formularorientierte Systeme
- Multiparadigmenorientierte Systeme

In fast allen Systemen lassen sich mehrere Ansätze erkennen, so dass es sich um multiparadigmenorientierte Systeme handelt. Wir werden uns die einzelnen Paradigmen im Folgenden genauer anschauen.

2.1. Steuerflussorientierte Systeme

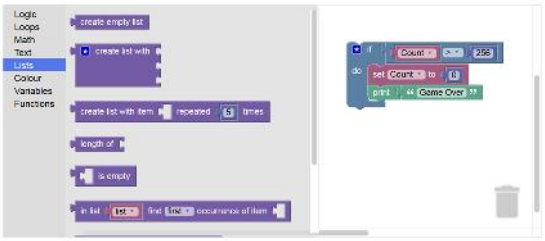
Diese Systeme entsprechen im Wesentlichen der imperativen Programmierung, ähnlich wie Sie es von den meisten textuellen Sprachen gewohnt sind. Die Ausführung von Anweisungen wird durch den Programmfluss mit einem Befehlszähler, der auf die nächste Anweisung zeigt, gesteuert. Meist handelt es sich daher um Anweisungssequenzen mit den üblichen Kontrollstrukturen wie if-then-else oder Schleifen. Aber auch Komponenten- und Transitionsnetzwerke gehören in diese Kategorie (an dieser Stelle sollen diese aber nicht vertieft werden).

Flussdiagramme sind ein Ihnen sicherlich bekanntes Beispiel für die Visualisierung von Anweisungssequenzen. Die Programmierung gleicht der textuellen Festlegung von Algorithmen, allerdings werden Anweisungen und Strukturen visuell aufbereitet. Dadurch werden Verschachtelungen und alternative Ablaufstränge klarer erkenntlich, da sie z.B. Seite an Seite dargestellt werden. Allerdings werden größere Programme schnell unübersichtlich. Diese Art von Programmdarstellung wird in der Regel für die Erläuterung von Algorithmen eingesetzt. Es gibt aber auch Systeme, die aus visuellen Diagrammen dieser Art ausführbare Programme erzeugen können.



Nassi-Shneiderman-Diagram

Das Programmieren mit Bausteinen (wie bei Scratch und EToys) ist im Prinzip an diese Art der Darstellung angelehnt. Wir finden diese Art der Programmfestlegung aber nicht nur für virtuelle Spielwelten sondern auch in der ernsthafteren Programmentwicklung. Aktuelle Projekte sind z.B. touchddevelop von Microsoft oder Blockly von Google. Blockly ist eine Javascript Bibliothek, die es Ihnen ermöglicht eine visuelle Programmiersprache für die Steuerung Ihrer eigenen Anwendungen einzusetzen.



https://www.touchdevelop.com und https://developers.google.com/blockly/

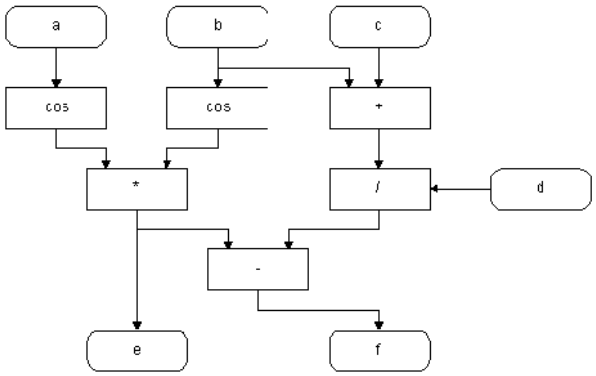
2.2. Objektorientierte Systeme

Wie bei der textuellen Programmierung handelt es sich hier um ein Paradigma, bei dem eine Abbildung der Welt mit Hilfe von Objekten modelliert wird. Während man bei der textuellen Programmierung seine Klassen und Objekte selbst festlegt, wird bei visuellen Systemen häufig auf fertige Komponenten zurückgegriffen, die zu komplexeren Einheiten zusammengestellt werden. Zwischen den Objekten können – ganz ähnlich wie beim Actors Konzept – Nachrichten ausgetauscht werden. Objekte besitzen eine Schnittstelle mit öffentlichen Eigenschaften, Nachrichtenkanälen oder Events. Events werden durch Benutzereingaben oder Zustandsänderungen ausgelöst und können zur Benachrichtigung anderer Objekte genutzt werden. Nachrichtenkanäle werden oft durch Pfeile visualisiert, wodurch die Programme schnell unübersichtlich werden. Die Programmdefinition ist meist ein dynamischer Prozess, d.h. Objekte werden konfiguriert und verändert. Bei EToys legt man z.B. Eigenschaften und Methoden dynamisch fest, durch Verschieben eines Objekts mit der Maus ändert sich das Programm. Es wird nicht streng zwischen Programmdefinition und Ausführung unterschieden, jede Änderung wird sofort sichtbar und aktiv.

2.3. Datenflussorientierte Systeme

Im Gegensatz zu steuerflussorientierten System gibt es bei datenflussorientierten Systemen keinen Befehlszähler, der die Reihenfolge der Anweisungsausführung festlegt. Die Verfügbarkeit von Daten definiert, welche Operation als nächstes ausgeführt wird. Wenn für eine Operation alle erforderlichen Daten vorliegen, dann kann diese ausgeführt werden. Dies ermöglicht den nebenläufigen Ablauf von Berechnungen, ohne dass man sich als Entwickler um die Synchronisation kümmern muss. Der Datenfluss legt fest, was als nächstes berechnet wird. Operationen liefern in der Regel neue Daten als Ergebnis, die dann wieder als Eingabewerte für weitere Operationen dienen.

Die Berechnung des arithmetischen Ausdrucks `e:=cos a * cos b and f:=e-(b+c) / d` könnte z.B. so aussehen:



Der Datenfluss wird visuell durch einen gerichteten Graphen festgelegt. Datenquellen, Datensenken und Operationen sind die Knoten des Graphen. Die Kanten dienen als Datenkanäle. Daten werden als (Nachrichten-)Tokens zwischen den Kanälen transportiert. Datenquellen, Datensenken und Operationen haben Eingangs- und Ausgangspunkte, die sich mit den Datenkanälen verknüpfen lassen. Eingangspunkte dürfen immer nur mit einem Datenkanal verknüpft sein, damit eindeutig ist, welches Datentoken als nächstes verwenden werden soll – es gibt also keine Wettlaufbedingungen. Ausgangpunkte können dagegen mit mehreren Kanälen verbunden sein, d.h. ein berechnetes Ergebnis kann als Eingabewert für mehrere weitere Operationen verwendet werden.

Datenquellen haben nur Ausgangspunkte, die mit mindestens einem Kanal verknüpft sein müssen. Sie können als Generatoren für Datenwerte (z.B. Zufallszahlen, Messwert, Zahlenfolgen) dienen oder konstante Werte enthalten. Eine Datensenke hat nur Eingangspunkte, wobei mindestens ein Kanal damit verbunden sein muss. Datensenken dienen oft der Ausgabe von Ergebnissen (z.B. Darstellung von Werten als Zahlen oder Diagramme). Eine Operation hat mindestens einen Eingangspunkt und mindestens einen Ausgangpunkt. Alle Eingangspunkte einer Operation müssen mit einem Datenkanal verknüpft sein; Ausgangspunkte müssen nicht zwingend mit Kanälen verbunden sein (man kann auch Mal ein Teilergebnis, an dem man nicht interessiert ist, ignorieren). Wenn eine Operation ausgeführt wurde, dann liegt an jedem Ausgangspunkt genau ein Ergebniswert vor. Das Vorliegen von Ergebniswerten kann dann die Ausführung weiterer Operationen anstoßen. Eine Operation wird nämlich ausgeführt, sobald alle erforderlichen Daten vorliegen. Somit wird die Ablaufreihenfolge alleine durch die Verfügbarkeit von Daten bestimmt.

Da Operationen immer lokal ausgeführt werden und auf keine globalen Datenwerte zugreifen (sondern nur auf Daten der Eingangspunkte), werden Seiteneffekte vermieden und Operationen können implizit nebenläufig ausgeführt werden, wenn für mehrere Operationen alle Daten vorliegen.

Mit LabView liegt seit 1983 ein sehr erfolgreiches datenflussorientiertes Programmiersystem vor, dass auch in größeren Projekten eingesetzt. LabView wird von der Firma National Instruments kommerziell vermarktet. Programme werden in der Programmiersprache „G“ mit Hilfe „virtueller Instrumente“ entwickelt. Ein virtuelles Instrument lässt sich wiederum aus virtuellen Instrumenten zusammensetzen. Dass von Instrumenten gesprochen wird, zeigt bereits, dass LabView häufig in der Mess-, Regel- und Automatisierungstechnik eingesetzt wird, und die Denkweise bei er Programmierung der Konstruktion realer Geräte entspricht. Zahlreiche fertige Komponenten ermöglichen eine schnelle Entwicklung. Die einzelnen virtuellen Instrumente entsprechen dabei den oben beschriebenen Operationen. Es gibt zudem Instrumente, die (kontrolliert) Seiteneffekte produzieren wie etwa das Steuern realer Maschinen. Zudem gibt es Knoten, mit denen sich auch

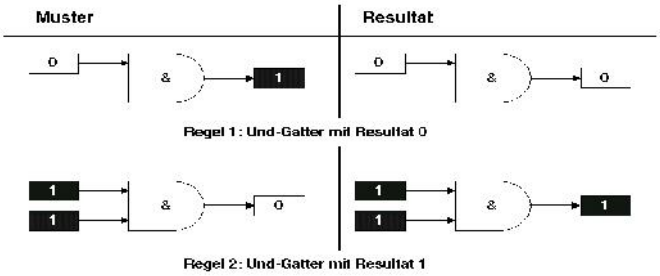
kontrollflussbasierte Funktionalität abbilden lässt, z.B. das wiederholte Ausführen einer Aufgabe wie bei einer Schleife.

Kontrollflussorientierte Systeme finden, wie gerade beschrieben, oft Einsatz zur Steuerung realer Maschinen. Dies ist bei LabView der Fall, aber auch bei anderen Systemen zur Steuerung von Robotern, sowohl in der Industrie wie für den Hausgebrauch (z.B. Lego Mindstorms, Fischertechnik Robotics). Sie spielen aber auch dort eine Rolle, wo die Verarbeitung von Daten und Informationen in mehreren Stufen erfolgt: beim Filtern von Nachrichten (z.B. Yahoo Pipes), Verknüpfung mehrerer Bildverarbeitungsoperationen (z.B. Blender) oder dem Analysieren verschiedener Datenquellen (z.B. IBM InfoSphere).

2.4. Regelorientierte Systeme

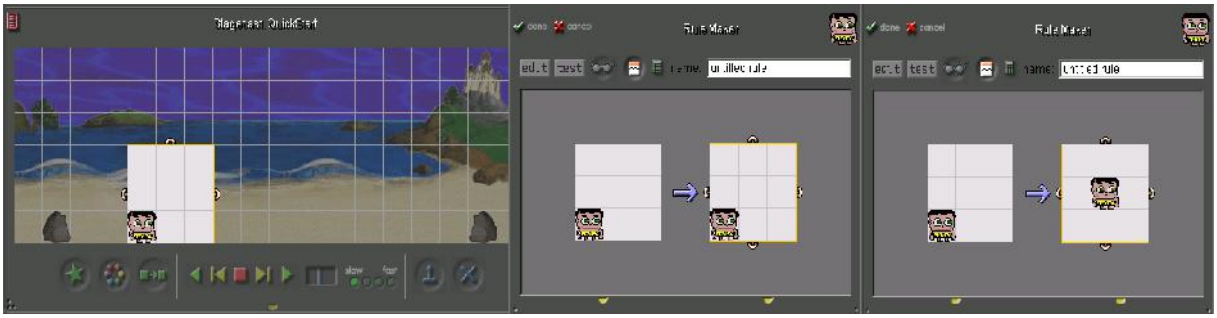
Diese Systeme beruhen auf einer Sequenz von Grafiktransformationen. Ein Programm wird durch eine Menge von Transformationsregeln festgelegt. Eine Transformationsregel besteht aus einer linksseitigen und einer rechtsseitigen Grafik. Die linke Seite definiert ein Suchmuster, die rechte Seite das Resultat, mit dem das gefundene Muster ersetzt wird. Ein solches Programm erwartet eine transformierbare Grafik als Eingabe. In dieser Grafik wird nun nach Teilgrafiken gesucht, die dem Suchmustern der Regeln entsprechen. Ist ein Muster gefunden, wird es durch das Resultat der Regel ersetzt. Dies geschieht fast wie beim „Alle vorkommen suchen und ersetzen“ einer Textverarbeitung, nur eben mit Grafiken. Das Programm terminiert wenn keine Muster mehr gefunden werden oder das Programm vom Benutzer unterbrochen wird. Problematisch bei diesem Paradigma ist, dass eventuell Muster mehrerer Regeln auf die aktuelle Grafik anwendbar sind. Daher ist es oft nötig, die Regeln zu priorisieren und zusätzliche Bedingungen einzuführen.

Hier ein Beispiel (im ChemTrains System) für eine Transformationsregel:



Erläuterung: Liegt beim &-Gatter irgendwo eine „0“ an, dann muss der Ausgang auch „0“ sein. Wird also in der aktuellen Transformationsgrafik irgendwo das Muster der ersten Regel gefunden (z.B. weil vorher eine andere Regel den Eingang von 1 auf 0 geändert hat), dann wird die Teilgrafik durch das Resultat ersetzt, so dass beim &-Gatter auch am Ausgang eine 0 anliegt.

Die für Schüler konzipierte Simulationsumgebung „StageCast“ arbeitet nach dem gleichen Prinzip. Grafische Zustände werden in Folgezustände überführt.

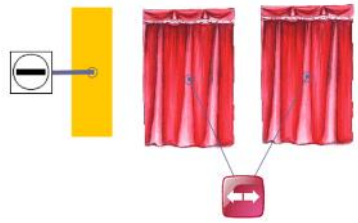


In der Breite haben sich regelbasierte Systeme jedoch nicht in der Praxis etabliert. Der Ansatz kann aber für Spezialanwendungen trotzdem interessant sein.

2.5. Constraintsorientierte Systeme

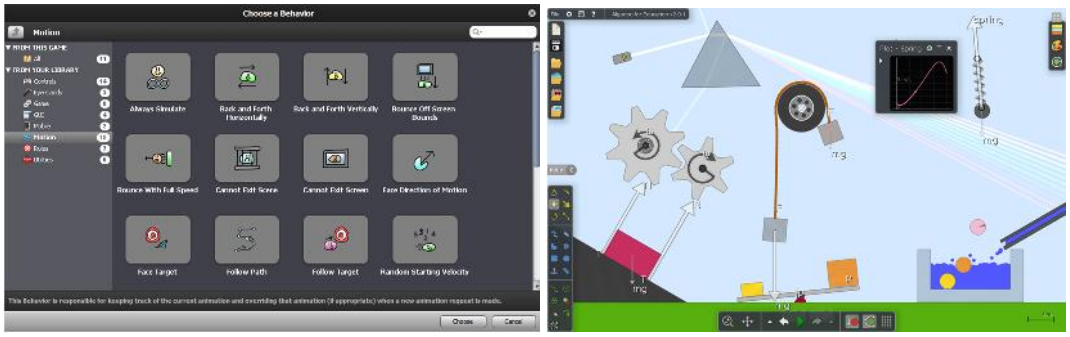
Constraintsorientierte Systeme legen bestimmte Einschränkungen und Beziehungen zwischen Objekten fest, die eingehalten werden müssen. Zum Beispiel könnte man für zwei nummerische Objekte festlegen, dass ein Objekt immer den doppelten Wert des anderen hält. Ändert man nun einen Wert, so wird der andere automatisch geändert. Dies bedeutet, dass nicht zwischen Ein- und Ausgabeobjekten unterschieden wird. Wenn ein Objekt sich ändert (z.B. durch Benutzereingaben), dann müssen sich automatisch andere Objekte ändern, damit die Beziehungen untereinander erhalten bleiben.

Constraints lassen sich auch für graphische Objekte gut festlegen, z.B. dass ein Objekt immer in der Mitte eines anderen Objekts positioniert ist, oder dass ein 3D-Charakter einem anderem Gegenstand folgt (Hinterherlaufen oder Ändern der Blickrichtung).



Constraints in moowinx

Es handelt sich hierbei um einen deklarativen Programmierstil, da nur Beziehungen zwischen Objekten festgelegt werden. Für die konsequente Einhaltung der Beziehungen ist dann das System zuständig. Man sagt also, was man haben möchte, aber nicht wie dies umgesetzt wird.



Stencyl (links) und Algodoo (rechts)

Als Spezialfall der constraintsorientierten Programmierung lässt sich das Festlegen von Verhaltensregeln für graphische Objekte nennen. Z.B. kann man für ein Objekt eine automatische Fortbewegungsregel festlegen, oder die Bewegungsfreiheit in eine Himmelsrichtung einschränken. Dieses Prinzip wird häufig von visuellen Entwicklungsumgebungen für Spiele („Game Maker“ wie Game Salad oder Stencyl) eingesetzt. Die Systeme sind meist ereignisgesteuert, d.h. beim Eintreten bestimmter Konstellationen (Events) finden die Regeln Anwendung. Ereignisse können Maus- oder Tastatureingaben, aber auch die Kollision mit anderen Objekten oder Bildschirmgrenzen sein. Die Regeln werden den visuellen Objekten zugewiesen, ohne dass man diese selbst programmieren muss. Hinter den Regeln steckt dann natürlich wieder regulärer Programmcode, der bei einigen Systemen auch selbst entwickelt werden kann.

2.6. Formularorientierte Systeme

Formularorientierte Systeme sind ein gutes Beispiel dafür, dass visuelle Programmierkonzepte auch in der Breite eingesetzt werden können, auch wenn uns dies gar nicht bewusst ist. Es handelt sich dabei nämlich um nichts anderes als Tabellenkalkulationen. Natürlich ist das „Programmieren“ mit einer Tabellenkalkulation nicht so mächtig wie mit einer Sprache wie C oder Java. Dennoch können Sie mit einer Tabellenkalkulation Rechenregeln festlegen. Die Beziehung zwischen Eingabewerten, Verarbeitungsregeln und Ausgabewerten wird dabei visuell festgelegt. Die Position der Zellen hat Einfluss auf das Ergebnis und Parameterlisten (also Zellenbereiche) werden visuell festgelegt.

2.7. Beispielorientierte Systeme

Hier wird eine Vorgehensweise programmiert, indem man dem System beispielhaft zeigt, wie es zu arbeiten hat. Makrorecorder zeichnen etwa eine Liste von Operationen aus, die anschließend wiederholt werden können, auch für andere Objekte. So lässt sich etwa in Bildbearbeitungsprogrammen wir Photoshop eine Reihe von Bearbeitungsschritten aufzeichnen. Anschließend können die gleichen Operationen für andere Bilder oder ganze Ordner mit Bilddateien als Stapelverarbeitung ausgeführt werden. Es gibt auch Rekorder, mit denen die Bewegungssequenz mit der Maus aufgezeichnet werden kann. Damit ist es zumindest theoretisch möglich, für jede beliebige Software eine Reihe von gleichen Operationen auszuführen. Somit lässt sich etwas ein zweites Dokument öffnen (z.B. eine Audiodatei) und danach die gleiche Reihenfolge der Arbeitsschritte wiederholen (z.B. das Regeln von Lautstärkeeinstellungen). Voraussetzung hierfür ist natürlich, dass die Benutzungsoberfläche immer gleich ist und somit z.B. ein Button oder ein Schieberegler immer an der gleichen Position bleibt.

3. Fazit

Es gibt sehr viele Paradigmen visueller Programmierung und noch mehr visuelle Programmiersprachen. Bis auf wenige Ausnahmen gibt es jedoch keinen durschlagenden Erfolg. Viele Systeme sind im wissenschaftlichen Umfeld als Experiment entstanden. Inzwischen gibt es aber auch einige Systeme, die für die ernsthafte Anwendungsentwicklung eingesetzt werden können. Dies gilt insbesondere für spezielle Bereiche wie die Entwicklung einfach Apps, Webseiten, E-Learningangebote, Multimediasysteme, Gerätesteuerung und Datenanalyse, sowie der Gestaltung von Simulationen und Spielen.

Häufig handelt es sich jedoch auch um Systeme, deren visuelle Repräsentation sehr einfach in eine textuelle Form überführt werden kann. Dies gilt insbesondere für die visuellen Programmblöcke der kontrollflussbasierten Systeme. Hier haben Anweisungen, Verzweigungen und Schleifen ein direktes Äquivalent in der textuellen Programmierung. So wird zwar erreicht, dass Anfänger- und Flüchtigkeitsfehler vermieden werden, da Syntax und Konfigurationsmöglichkeiten fest vorgegeben sind. Das algorithmische Denken und das Begreifen von Zusammenhängen sind hier aber genauso erforderlich wie bei textuellen Sprachen. Gleichzeitig handelt man sich einige Probleme ein. So stehen höhere Programmierkonzepte wie Vererbung, Polymorphie, funktionale Programmierung meist nicht zur Verfügung. Die eigentlichen Konzepte der Programmierung werden sogar häufig versteckt hinter der visuellen Aufbereitung, so dass ein tiefergehendes Verständnis über die Abläufe häufig fehlt. So lassen sich zwar schnell einfache Ergebnisse erzielen, umfangreichere Projekte sind oft nur eingeschränkt möglich. Dies liegt auch an den beschränkten Abstraktionsmöglichkeiten und der schweren Wartbarkeit. Ein weiteres Problem liegt darin, dass graphische Notationen oft mehrdeutig sind, z.B. kann das Verschachteln von Kästen mehrere Bedeutungen haben. Der begrenzte Bildschirmplatz scheint die visuelle Programmierung ebenfalls zu erschweren. So beschreibt das Deutsch-Limit, dass durch die begrenzte Anzahl gleichzeitig sichtbarer Elemente das Definieren

komplexerer Systeme sehr schwer wird (Peter Deutsch stellte einst die Frage, wie man wohl mit 50 gleichzeitig sichtbaren Primitiven ein Betriebssystem entwickeln möchte).

Die visuelle Notation eignet sich besonders dann, wenn die Anwendung selbst viele graphische Objekte enthält, da eine direkte Repräsentation der Komponenten möglich ist. Zudem eignen sich graphische Darstellung für die Wahrnehmung von Zusammenhängen und Strukturen (z.B. Alternative Ablaufzweige, Verschachtelungen, Kollision). Textuelle Darstellungen eignen sich dagegen besonders gut für sequentielle Abläufe, Abstraktionen, Referenzen die Umsetzung komplexerer Systeme.