Algoria: Tablet-PC Anwendung für den Informatikunterricht

Algoria ist eine neuartige, stiftbasierte Tablet-PC Anwendung für den Informatikunterricht. Sie ist in der Lage im eng begrenzten Kontext der algorithmischen Datenstrukturen typische Skizzen von Arrays, Listen, Bäumen und Graphen direkt während des Skizzierens zu erkennen, in entsprechende Datenstrukturen im Hauptspeicher abzubilden und Algorithmen darauf anzuwenden. Die von den Algorithmen erwirkten Anpassungen der Datenstrukturen werden in Form von automatisch generierten Animationen dargestellt. In diesem Artikel gehen wir hauptsächlich auf den Aufbau der Software und auf die eingesetzte GUI-Technologie WPF ein. Zudem zeigen wir, wie mit WPF eine konsequente Trennung zwischen GUI-Design und -Verhalten erreicht werden kann.

Raphael Schweizer, Christoph Stamm, Beat Walti | christoph.stamm@fhnw.ch

2002 kamen die ersten Laptops mit berührungssensitivem Bildschirm, so genannte Tablet-PCs[1], auf den Markt. Seit den vielen unterschiedlichen Anpreisungen und Lobeshymnen sind die Geräte und die dazu notwendige Software ständig weiterentwickelt worden. Mittlerweile haben sich auch Apple und RIM mit ihrem *iPad* bzw. *PlayBook* der Sache angenommen und es ist absehbar, dass der von Windows dominierte Tablet-PC Markt dadurch aufgefrischt wird. Eine solche Auffrischung kann kaum schaden, denn bis heute hat sich der Tablet-PC – zumindest im europäischen Raum – nur in wenigen vertikalen Märkten durchgesetzt, etwa in Krankenhäusern oder in Industriebetrieben zur Steuerung und Überwachung komplexer Abläufe. Konsumenten hingegen haben den Tablet-PC bis heute nicht wirklich als Laptop-Ersatz angenommen. Ist der Tablet-PC demnach eine Fehlentwicklung oder erfüllen die Geräte die in sie gesteckten Erwartungen nicht?

In einem von der Haslerstiftung[2] geförderten Projekt[3] wollen wir exemplarisch aufzeigen, dass Tablet-PCs im schulischen Umfeld ihre Berechtigung haben, dass die Schule wie so oft als Technologiewegbereiter dienen kann und dass die Schlüsseltechnologie der Tablet-PCs – die Handschrifterkennung – mittlerweile die notwendige Reife erlangt hat, um für eine Vorlesungsmitschrift zu genügen. Im Gegensatz zu anderen Forschungsprojekten, welche den allgemeinen Nutzen von Tablet-PCs im Lernprozess analysieren [AC10, AU10, COP09] oder einen gezielten Umgang mit Tablet-PCs im technischen Unterricht aufzeigen [BBC10, KFE07], wollen wir zeigen, dass Tablet-PCs nicht nur einzelne negative Begleitumstände von Laptops in Schulzimmern eliminieren können, sondern darüber hinaus eine neuartige und in der Informatik bisher selten erlebte Form des enaktiven Erlernens (durch Schülerhandlungen) von rein abstrakten Abläufen mit entsprechender Software ermöglichen. Das Arbeiten mit Stift und Touch-Screen erlaubt neben dem vereinfachten Zusammenarbeiten in Gruppen einerseits den Einsatz virtueller Leinwände und anderseits eine eng verknüpfte Kombination des Skizzierens und gleichzeitigen Simulierens. Diese beiden Stärken wollen wir mit unser neu entwickelten Lernsoftware *Algoria* für Informatik-Dozierende und -Studierende exemplarisch demonstrieren und im Unterrichtsfach „Algorithmen und Datenstrukturen“ auch austesten.

Algoria ist in der Lage im eng begrenzten Kontext der algorithmischen Datenstrukturen typische Skizzen von Arrays, Listen, Bäumen und Graphen während der Erstellung zu erkennen und zu prozessieren. Die Art des Prozessierens geht dabei über die bekannte Skizzenerkennung wie in [PH08, Rbe06, Ham07] beschrieben hinaus und beinhaltet bei uns auch die abstrakte Abbildung der Datenstruktur im Speicher und schafft dadurch die Grundlage zur Simulation und Animation von Algorithmen. Gerade die Animation der Algorithmen, wie wir sie von unzähligen Applets im Internet kennen, z.B. [Muk], ist oft ein hilfreicher Bestandteil fürs Verständnis. Damit sprengt Algoria auch das Spektrum von Anwendungen, welches [KHPC08] mit ihrem System zur vereinfachten Erstellung von Tablet-PC Anwendungen anvisieren. Kurz gesagt, Algoria verschmelzt ausgeklügelte Skizzenerkennung mit automatisch generierter, eindrucksvoller Animation.

In diesem Artikel beschreiben wir zuerst ein paar Grundkonzepte, die bei der Entwicklung von Tablet-PC-Anwendungen beachtet werden sollen, führen dann in die Skizzenerkennung ein und lei-

1 Heutzutage wird zwischen reinen Tablets und Convertibles unterschieden. Wir verwenden hier den Begriff Tablet-PC für beide Ausprägungen.

2 www.haslerstiftung.ch

3 Algoria: Tablet-PCs im Informatikunterricht; 2009 bis 2011

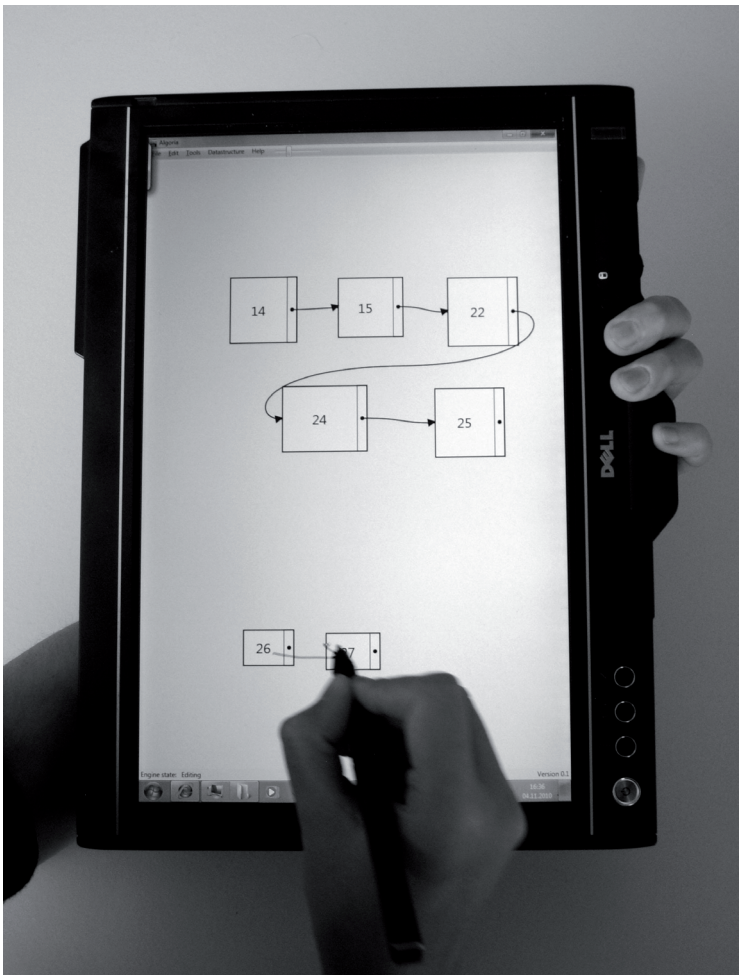

Abbildung 1: Algoria im praktischen Einsatz. Die Benutzerin verbindet die soeben gezeichneten Listenelemente „26“ und „27“ mit einem Pfeil.

ten schliesslich zur Architektur von Algoria und zu den verwendeten Technologien über. Dabei richten wir unser Augenmerk auf das in Algoria verwendete GUI-Framework WPF. Mit einer einfachen Beispielanwendung zur Listendarstellung zeigen wir exemplarisch auf, wie WPF eine konsequente Trennung zwischen GUI-Design und -Verhalten ermöglicht. Den Bericht schliessen wir mit einem kurzen Ausblick auf das weitere Projektvorgehen ab.

Enge Platzverhältnisse

Tablet-PCs haben üblicherweise kleine Bildschirme (ca. 11") mit einer oft bescheidenen Auflösung. Daher ist ein haushälterischer Umgang mit der Bildschirmfläche angebracht. Auf unnötig breite Menübalken, wie man sie beispielsweise vom *Ribbon-Control* aus *Microsoft Office 2007* [4] kennt, sollte wenn möglich verzichtet werden. In Algoria versuchen wir generell auf *Toolbars* und Menübalken zu verzichten. Dies führt zu einem spartanischem GUI, aber genau das ist schliesslich unser Ziel: viel Platz für das Wesentliche. Dabei stellt sich jedoch die Frage, wie man trotz kargem GUI sämtliche Funktionalität bequem und per Stift intuitiv und mit kurzen Wegen erreichbar machen kann.

Die aufgeworfene Fragestellung ist in verschiedensten Studien bereits untersucht worden, z.B. in [GW00, Hop91, KB94]. In all diesen drei Ansätzen wird ein Interface vorgeschlagen, welches sich in runder Form um die Stiftposition herum anordnet, so dass möglichst wenig Strecke mit dem Stift zurückgelegt werden muss. Im Unterschied zum Ansatz in [GW00] möchten wir nicht auf Klicks verzichten, dafür aber auf das bewährte Popup-Menü bei Rechtsklick[5], welches zur Reduktion der Handbewegung und zur Ausnutzung des Kontexts eingeführt wurde. Obwohl das Popup-Menü im Umgang mit der Maus eine sehr wertvolle Hilfe darstellt, so ist der Rechtsklick mit einem Stift eher schwierig vorzunehmen, da der zu betätigende Stiftknopf nur mühsam gedrückt werden kann. Viele der kleinen Stiftbewegungen werden durch den Zeigefinger der Schreibhand gesteuert. Wird der Zeigefinger nun zur Betätigung des Rechtsklicks abkommandiert, führt die Schreib-

4 Das GUI in Office lässt sich auch so konfigurieren, dass dieser Platz nicht oder nicht permanent beansprucht wird.

5 Apple verzichtete ursprünglich auf eine Maus mit mehreren Tasten und somit auch auf das Kontextmenu bei Rechtsklick.

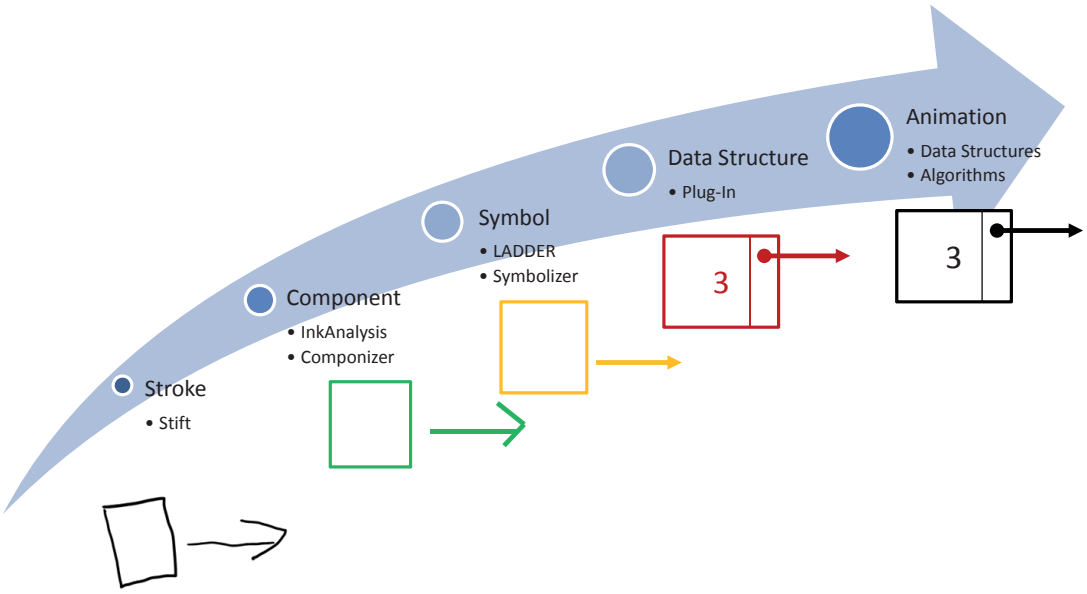

Abbildung 2: Der Ablauf vom Strich über die Datenstruktur bis hin zum animierten Algorithmus

hand dennoch reflexartig eine Stiftbewegung aus, was zu einer leicht veränderten Bildschirmposition und somit zu einem Rechtsklick an einer unerwünschten Position führen kann. Durch ein Redesign des Stiftes könnte dieser Nachteil allenfalls aus der Welt geschaffen werden.

Neben dem kleinen Stiftknopf unterstützt Windows auch die Emulation eines Rechtsklicks durch einen zeitlich verlängerten Linksklick. Dieser Ansatz löst zwar das Problem der ungenügenden Präzision mit dem Stiftknopf, erzeugt aber ein anderes: Beim Popup-Menü mit der Maus hat man sich daran gewöhnt, dass dieses Menü unmittelbar nach dem Rechtsklick auftaucht und somit kann die Hand sofort mit einer seitlichen Bewegung zur Auswahl des Menüeintrags beginnen. Eine Abkehr von diesem zeitlichen Verhalten erschwert die Arbeit mit dem Stift unnötig, so dass eine Umstellung und vor allem ein häufiger Wechsel zwischen Maus und Stift sehr unangenehm sind. Aus den zuvor genannten Gründen propagieren wir den Verzicht auf den Rechtsklick in Tablet-PC-Anwendungen und untersuchen sinnvolle Alternativen.

Vom Strich zur animierten Datenstruktur

Das Zeichnen einer Datenstruktur beginnt ganz einfach mit einem Strich (Abb. 2). So ein Strich ist eine Punktekette, welche verschiedene Formen annehmen kann: Liniensegment, Kreisbogen, Ellipse, Polygon, usw. Aus den Strichen werden geometrische Komponenten eines Symbols abgeleitet, wobei jedes Symbol aus endlich vielen Komponenten besteht. Beispielsweise besteht das Symbol „Pfeil“ aus einer Pfeilspitze und einem Schaft. Zusammen bestehen sie aus mehreren geometrischen Komponenten (z.B. drei Liniensegmente oder ein Liniensegment und ein Dreieck).

Welche Komponenten für ein Symbol benötigt werden und in welchen Beziehungen diese Komponenten zueinander stehen, wird in ausgelagerten LADDER-Dateien spezifiziert [HD03, Ham07b, Sch10]. Dadurch erreicht man eine klare Auftrennung zwischen abstraktem Symbol und seiner grafischen Darstellung. Diese Aufteilung ist analog zur Definition von abstrakten Schriftzeichen in Codierungsstandards[6] und deren konkreter grafischen Umsetzung in Form von Schriftarten (Fonts). Eine solche Aufteilung drängt sich aus zweierlei Gründen auf: erstens weil es unterschiedliche Darstellungsarten für die gleichen Datenstrukturen gibt und zweitens weil die gezeichnete Darstellung (Symboleingabe) nicht unbedingt der grafischen Symbolausgabe entsprechen muss. Gerade dieser zweite Grund erhebt das hier vorliegende virtuelle Zeichnen auf eine höhere Abstraktionsstufe verglichen mit dem physischen Zeichnen z.B. auf Papier.

Eine Datenstruktur besteht aus einer Menge gleichartiger Elemente, die untereinander (semantisch) verknüpft sind. In Abbildung 2 sehen wir beispielsweise ein Element einer einfach verketteten Liste. Ein solches Element kann jedoch auch als vollständige, 1-elementige Datenstruktur betrachtet werden. Unter der Voraussetzung, dass die Datenstrukturen in der Lage sind, sich mit anderen Datenstrukturen zu verschmelzen, reicht es zur Beschreibung der Datenstruktur ein einzelnes Datenstrukturelement als Kombination von Symbolen oder als Symbol höherer Ordnung zu beschreiben. Diese Beschreibung erfolgt nach dem gleichen Prinzip wie die Beschreibung eines Symbols selber. Gegenüber dem Symbol legt die Datenstruktur jedoch zusätzlich noch das Verhalten ihrer Symbole fest. Typische Beispiele

6 Unicode Standard, www.unicode.org

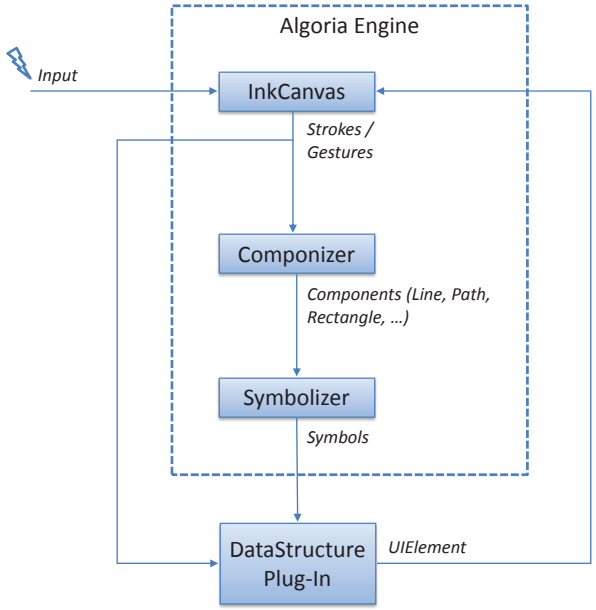

Abbildung 3: Datenflusses in Algoria

für Verhalten sind das Verschmelzen von Datenstrukturen, die Akzeptanz von Texteingabe oder die Bereitstellung von Geometriemanipulatoren. Anders ausgedrückt, die Datenstruktur liefert die Semantik zu einem Symbol. Der Pfeil in unserem Listenelement wird dann beispielsweise zur Verkettung zweier Listenelemente.

Aus den gezeichneten und erkannten Symbolen bzw. Datenstrukturelementen kann nun eine gesamtheitliche Datenstruktur im Hauptspeicher des Systems aufgebaut werden. Damit ist die Grundlage zur Ausführung von Algorithmen auf dieser Datenstruktur gelegt. Welche Algorithmen pro Datenstruktur jedoch angeboten werden, ist individuell und lässt sich über einen eingebauten Plug-In-Mechanismus von aussen erweitern. Normalerweise ist die Ausführung eines Algorithmus für einen Betrachter nur indirekt über die Daten- oder Strukturveränderung ersichtlich. Da jedoch längst nicht alle Algorithmen die Struktur verändern und viele Datenveränderungen auf den ersten Blick chaotisch anmuten, ist eine visuelle Darstellung des algorithmischen Ablaufs oft wünschenswert. Wir sprechen hierbei von einer Algorithmen-Animation. In den meisten eindrucksvollen Algorithmen-Animationen wird für die eigentliche Animation ein Vielfaches an Programmcode des ursprünglichen Algorithmus benötigt. Dadurch wird der animierte Algorithmus zu einem aufgebauschten Kunstprodukt, das nicht mehr einfach nur abläuft, sondern auf vielfältigste Art sich selbst (z.B. in Form von Quellcode) und seinen Ablauf visualisiert. Das Problem dabei ist, dass der Animationscode nicht aus dem Programmcode automatisch erzeugt werden kann, was zur Folge hat, dass zum Algorithmus noch aufwendig hergestellter Animationscode mitgeliefert werden muss. Eine minimale Form der Algorithmenanimation verzichtet grösstenteils auf solch

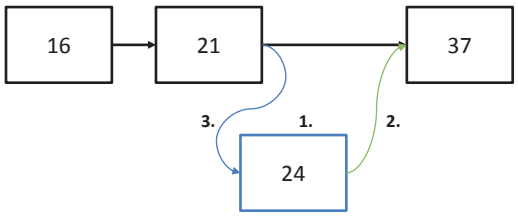

Abbildung 4: Verschiedene Zustände während einer Einfügeoperation

zusätzlichen Animationscode und versucht stattdessen, aus dem Programmcode des Algorithmus den Animationscode automatisch zu generieren. In Algoria gehen wir diesen zweiten Weg.

Algoria im Überblick

In Abbildung 3 sehen wir den hauptsächlichen Datenfluss in Algoria. Die von der Benutzerin eingegebenen Striche werden auf dem *InkCanvas* erfasst und dahingehend untersucht, ob es sich dabei um vordefinierte Gesten handelt. Erkannte Gesten und unbekannte Strichfolgen werden anschliessend einer allenfalls bereits vorhandenen Datenstruktur übergeben, welche entscheidet, ob sie die Eingabe verarbeiten kann, was zum Beispiel bei handgeschriebenem Text innerhalb eines Listenelementes zum Tragen kommt. Für den Fall, dass keine Datenstruktur für die Eingabe zuständig ist oder eine zuständige Datenstruktur die übergebenen Striche nicht verarbeitet, versucht der *Componizer* die Eingabe zu neuen Komponenten zu verarbeiten. Die Komponenten werden dann alleine oder mit anderen Komponenten zusammen durch den *Symbolizer* zu Symbolen kombiniert und der Datenstruktur übergeben. Schliesslich sorgt die Datenstruktur dafür, dass das skizzierte Symbol auf dem *InkCanvas* durch eine verschönerte und für die Datenstruktur typische Repräsentation ersetzt wird.

Die drei Module *InkCanvas*, *Symbolizer* und *Componizer* bezeichnen wir als *Algoria Engine*. Ausserhalb dieser Engine befinden sich individuelle Plug-Ins, welche die unterschiedlichen Datenstrukturen kapseln und mit der Engine über vorgegebene Schnittstellen interagieren. Wie ein solches Zusammenspiel zwischen Plug-In und Engine im praktischen Ablauf aussehen kann, beschreiben wir hier für das konkrete Beispiel einer einfach verketteten Liste beim Einfügen eines neuen Listenelementes.

Nehmen wir an, dass bereits eine Liste mit drei Elementen gezeichnet und mittels Pfeilen verkettet worden ist (Abb. 4). Zum Einfügen eines neuen Elementes „24“ zeichnen wir es in der Nähe der Einfügeposition und verbinden es anschliessend mit einem neuen Pfeil zum bereits vorhandenen Element „37“. In einem dritten Schritt zeichnen wir einen neuen Pfeil zwischen dem Element „21“ und dem neuen Element und setzen so das neue

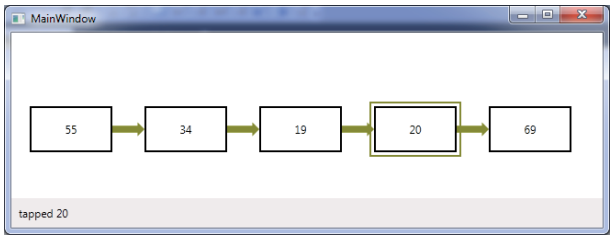

Abbildung 5: Einfache WPF-Beispielapplikation

Element als direkten Nachfolger von „21“. Gleichzeitig entfernt Algoria die bisherige Verbindung zwischen „21“ und „37“. Dieser beschriebene Ablauf entspricht der üblichen Vorgehensweise beim Einfügen eines neuen Elementes in eine verkettete Liste. In Algoria tritt jedoch dabei das Problem auf, dass nach dem zweiten Schritt zwei separate Listen existieren, welche beide eine gemeinsame Fortsetzung (Listenelement „37“) haben. Dieser Umstand erschwert generell eine konsistente Visualisierung der Listen und muss mit passenden Massnahmen (z.B. kongruente visuelle Überdeckung der gemeinsamen Listenteile) gelöst werden.

Wie eingangs beschrieben, werden vordefinierte Strichfolgen vom *InkCanvas* als Gesten erkannt. Solche Gesten erlauben eine intuitivere Bedienung eines Touchscreens. In unserem Beispiel mit der Liste liesse sich etwa ein Listenelement durch eine mindestens dreifache, horizontale Strichfolge entfernen. Diese Geste ist der Benutzung eines Radiergummis nachempfunden.

Verwendete Technologien

Neben dem Zeichnen von Symbolen spielt auch die stiftbasierte Texteingabe und die dazugehörige Texterkennung eine wichtige Rolle in Algoria. Die Texterkennung wird nicht von Algoria selber erledigt, sondern an das Tablet-PC API von Microsoft delegiert. Das Tablet-PC API, welches auch schlecht lesbaren Text, verschiedene Sprachen und einfachste Geometrien und Gesten erkennen kann, ist seit der Windows XP Tablet-PC Edition im Betriebssystem integriert. In anderen Betriebssystemen wie Linux oder Mac OSX gibt es bislang ähnliche Erweiterungen nur von Drittanbietern, was üblicherweise dazu führt, dass die gewünschte Funktionalität nicht auf allen Installationen vorhanden ist oder sich inkonsistent verhält.

Auf das Tablet-PC API von Windows kann nativ oder auch mittels .Net Code zugegriffen werden. Infolge der Aktualität des .Net Frameworks und der verbesserten Wartbarkeit des Codes haben wir uns für eine Entwicklung in C# entschieden. Als dazu passende GUI-Technologien bieten sich vor allem *WinForms* und *Windows Presentation Foundation* (WPF) an. WPF ist die neuere der beiden Technologien und kann als Ersatz von *WinForms* angesehen werden. WPF überzeugt vor allem durch seine konsequente Trennung von Darstellung und Verhalten und sorgt dafür, dass sich GUI-Elemente auf einfache Art (bidirektional) an ein Datenmodell binden lassen. Gerade das einfache Binding zwischen Datenmodell und GUI erleichtert auch die Animation von Algorithmen, da die Algorithmen lediglich auf dem Datenmodell operieren müssen.

Windows Presentation Foundation

Mit Hilfe einer einfachen C# WPF-Applikation (Abb. 5) wollen wir die konsequente Trennung zwischen Darstellung und Verhalten aufzeigen. Diese Beispielapplikation soll einerseits verschiedene Techniken von WPF demonstrieren und andererseits deren Nutzen oder Schwierigkeiten aufdecken. Ähnlich wie in ASP.Net werden pro GUI-Komponente zwei separate Dateien angelegt: Das Design wird in einer XML-ähnlichen Sprache, einer XAML-Datei (*eXtensible Application Markup Language*) beschrieben und das Verhalten in einer so genannten *Code-Behind* Datei in C# implementiert.

In XAML wird der hierarchische und logische Aufbau des GUIs (*LogicalTree*) oder der visuelle Aufbau einer Komponente (*VisualTree*) beschrieben. Im *LogicalTree* sind all jene abstrakten GUI-Komponenten enthalten, die man als komplette, einzelne Bausteine verwendet, wie beispielsweise eine Schaltfläche oder ein Textfeld. Der *VisualTree* beinhaltet all jene visuellen Bestandteile, welche tatsächlich gezeichnet werden. Bei einer Schaltfläche ist dies nebst einem Rahmen auch sein Hintergrund, der Text auf der Schaltfläche, ein Schatten, ein Mauseffekt usw. Der logische Baum kann also als generalisierter Baum aller visuellen Komponenten gesehen werden.

Unsere Beispielapplikation soll in der Lage sein, eine veränderbare Liste darzustellen. Sie besteht im Wesentlichen aus zwei Komponenten:

- einem *InkCanvas* und
- einer beispielhaften Implementierung einer Liste.

Das *InkCanvas* ermöglicht es uns, mittels Stift oder Maus auf der gesamten Oberfläche der Applikation zu zeichnen. Das Gezeichnete wird analysiert und falls es keiner der vier nachfolgenden Gesten entspricht, verbleibt es als Strichfolge auf der Oberfläche:

- *ScratchOut* (mindestens dreifaches horizontales links/rechts ziehen auf einer Höhe): es werden all jene Listenelemente gelöscht, die während der Geste berührt werden;
- *Left* (schnelles nach links ziehen): das Datenmodell wird reinitialisiert;
- *Right* (schnelles nach rechts ziehen): löscht alle nicht verarbeiteten Striche vom *InkCanvas*;
- *Tap* (antippen): setzt den Fokus auf ein Listenelement oder entfernt ihn wieder.

Während die beiden Gesten *Left* und *Right* nicht von der Datenstruktur selber verarbeitet werden,

```
<DataTemplate X:Key="ListItemTemplate">
  ...
  <Border Width="90" Height="50"
    BorderBrush="Black" BorderThickness="2"
    Background="White" >

    <i:Interaction.Triggers>
      <local:GestureTrigger Gesture="ScratchOut" >
        <local:DeleteListFieldAction />
      </local:GestureTrigger>
      <local:GestureTrigger Gesture="Tap">
        <local:ToggleFocusAction />
      </local:GestureTrigger>
    </i:Interaction.Triggers>

    <TextBlock Text="{Binding Value}" IsHitTestVisible="False"
      HorizontalAlignment=”Center”
      VerticalAlignment=”Center”/>
  </Border>
  ...
</DataTemplate>
```

Listing 1: Design eines Listenelementes

sondern üblicherweise von einem Controller, da die visualisierte Datenstruktur während der Gesten gar nicht berührt werden muss und somit auch nicht die Ereignisse erhält, werden hingegen die beiden anderen Gesten (*ScratchOut* und *Tap*) direkt von der Datenstruktur bzw. deren Visualisierung verarbeitet. Dabei zeigt sich ziemlich gut, wie die Trennung von Verhalten und Darstellung zu verstehen ist.

Listing 1 implementiert das GUI eines Listenelements, bestehend aus einem Rahmen und einem Textblock. Das Verhalten des Listenelementes wird durch die darauf anwendbaren Gesten (*ScratchOut* und *Tap*) und die damit ausgelösten Aktionen beschrieben. Diese Beschreibung wird mittels *Trigger* und *Action* angehängt und ist somit weder im Datenmodell noch direkt in der Visualisierung implementiert. Bei einer *ScratchOut*-Geste besteht das grundsätzliche Problem, dass sie infolge ihrer räumlichen Ausdehnung mehrere GUI-Komponenten erfassen kann. Falls mehrere Listenelemente gleichzeitig gelöscht werden sollen, dann ist dieses Verhalten durchaus wünschenswert. Wenn aber ein Listenelement, wie in unserem Fall, aus mehreren GUI-Komponenten besteht, sollte darauf geachtet werden, dass nur eine einzige Delete-Action für das gesamte Listenelement ausgelöst wird. Durch die Deaktivierung des *HitTests* auf einem Teil der Komponenten (im Beispiel auf dem *TextBlock*) kann dieser Effekt elegant erzielt werden. Diese Deaktivierung hat aber auch Einfluss auf die punktuelle *Tap*-Geste, welche üblicherweise im Innern des Rahmens und somit auch innerhalb des Textblocks erfolgt. Durch die Deaktivierung wird der Textblock beim Berührungstest nicht beachtet. Stattdessen empfängt der dahinterliegende Rahmen die *Tap*-Geste und verarbeitet sie oder leitet sie im *VisualTree* an sein umgebendes Element weiter. Eine Geste ist ein so genannter *RoutedEvent*, der so lange im *VisualTree* nach oben wandert, bis er in einem passenden *Gesture-Handler* verarbeitet wird. Dieses Konzept ist weitaus praktikabler für GUIs als das *Observer-Pattern*, bei dem sich Objekte für den Erhalt von Ereignissen speziell registrieren müssen.

Die Darstellung der gesamten Liste beschränkt sich auf wenige XAML-Zeilen, wie in Listing 2 ersichtlich ist. Das Property *ItemTemplate* definiert die Darstellung eines einzelnen Listenelementes (wie in Listing 1 gezeigt), *ItemsSource* bindet auf eine Instanz der Liste („MyList“) und *ItemsPanel* definiert das Layout der Liste. Die Darstellung eines einzelnen Listenelementes ist also ganz unabhängig von der Anordnung der Listenelemente und beide Teile können selber gestaltet werden. In unserem Beispiel verwenden wir nur für ein Listenelement ein eigenes Template. Für das Layout aller Listenelemente der Liste greifen wir auf ein bereits vorhandenes *StackPanel* zur horizontalen Anordnung zurück.

```
<ItemsControl ItemTemplate="{StaticResource ListItemTemplate}"
  ItemsSource="{Binding MyList}">
  <ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel Orientation="Horizontal" />
    </ItemsPanelTemplate>
  </ItemsControl.ItemsPanel>
</ItemsControl>
```

Listing 2: Design der Liste

Ein weiteres WPF Element, welches wir an dieser Stelle hervorheben wollen, ist der *Adorner*. Ein *Adorner* kann man sich als Dekoration für eine GUI-Komponente vorstellen, die relativ zur dekorierten GUI-Komponente platziert wird. In Abbildung 5 ist das Listenelement „20“ beispielsweise mit einem *BorderAdorner* hervorgehoben. *Adorner* werden in einem eigenen Layer im Vordergrund gezeichnet und können dadurch nicht durch GUI-Komponenten überdeckt werden. Da allerdings mehrere *Adorner* gleichzeitig aktiv sein können, kann es durchaus vorkommen, dass mehrere *Adorner* einander überdecken. Obwohl *Adorner* Teil der Visualisierung sind, werden sie in C# und nicht in XAML implementiert. Dies hängt damit zusammen, dass *Adorner* primitive *FrameworkElemente* sind und die Trennung zwischen XAML- und C#-Code erst auf einer höheren Abstraktionsebene hinzukommt.

In unserer Beispielanwendung verwenden wir den *BorderAdorner* aus Listing 3, um den Fokus eines Listenelementes zu visualisieren. Im Konstruktor wird ein Rechteck vorbereitet, welches dann später in der *ArrangeOverride*-Methode ausgerichtet und (indirekt von WPF) gezeichnet wird. Damit WPF erkennt, dass der *Adorner* etwas zu visualisieren hat, muss das Rechteck als *VisualChild* registriert werden. Dies geschieht mittels der beiden *Properties Children* und *VisualChildrenCount* sowie der Methode *GetVisualChild(int index)*.

Plug-In Datenstruktur

In Algoria bestehen alle unterstützten Datenstrukturen aus einem umfangreichen Set von Programmkomponenten. Diese Funktionen und Controls werden pro Datenstruktur zusammengefasst und in ein oder mehrere separate Plug-Ins ausgelagert. Eine solche Abtrennung von der Algoria Engine erlaubt eine separate Entwicklung

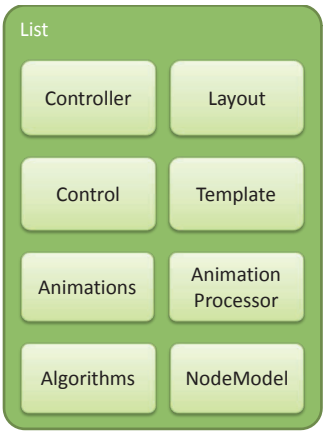

Abbildung 6: Komponenten einer Listenimplementierung

und somit auch eine erleichterte Erweiterbarkeit. Während die Entwicklung einer neu zu unterstützenden Datenstruktur einiges an Wissen über Algoria voraussetzt, ist die Entwicklung und Integration von neuen Algorithmen für die bestehenden Datenstrukturen auch für Entwickler ohne spezifisches Algoria-Wissen möglich.

Das Plug-In der Datenstruktur „List“ besteht aus einem *Controller*, einem *Layout*, einem *Control*, einem *Template*, einem *NodeModel* und einigen Animationen (Abb. 6). Der *Controller* instanziiert und initialisiert für jedes übergebene *Node*-Symbol ein neues *NodeModel* und generiert dazu passend das *Control*, welches die Visualisierung übernimmt. Das *Control* wendet dafür das *Template* auf das *NodeModel* an und generiert so pro *NodeModel* eine eigenständige Visualisierung, die dann wiederum mittels *Layout* innerhalb des Listen-*Controls* an der entsprechenden Stelle gezeichnet wird. Bei animierten Algorithmen entscheidet der *AnimationProcessor*, welche Teilmenge der vom Algorithmus automatisch generierten Ereignisse wirklich animiert wird. Zu-

```
public class BorderAdorner : Adorner {
  protected VisualCollection Children { get; set; }
  protected override Visual GetVisualChild(int index) { return Children[index]; }
  protected override int VisualChildrenCount { get { return Children.Count; } }

  private Rectangle rect;

  public BorderAdorner(UIElement adornedElement) : base(adornedElement) {
    Children = new VisualCollection(this);
    rect = new Rectangle {
      Stroke = new SolidColorBrush(Colors.DarkOliveGreen),
      StrokeThickness = 3
    };
    Children.Add(rect);
  }
  protected override Size ArrangeOverride(Size finalSize) {
    FrameworkElement fe = AdornedElement as FrameworkElement;
    if (fe != null) {
       rect.Arrange(new Rect(0, 0, fe.ActualWidth, fe.ActualHeight));
    }
    return base.ArrangeOverride(finalSize);
  }
}
```

Listing 3: Ein einfacher Adorner, welcher einen Rahmen um ein bestehendes Listenelement zeichnet.

```
public interface IDatastructureController {
  BitmapSource Icon { get; }
  string Name { get; }
  string Description { get; }

  bool IsLinked { get; }
  IEnumerable<string> SymbolsOfInterest { get; }

  IDatastructure Create(ISymbolAnalysis sa);
  bool TryMerge(ref IDatastructure first, ref IDatastructure second, ISymbolAnalysis link);
}
```

Listing 4: Der Einstiegspunkt in das Plug-In

dem wählt er die passenden Animationen aus und spielt diese ab.

Im folgenden Abschnitt gehen wir auf den Controller, das Datenmodell und das Layout noch etwas genauer ein.

Controller, Datenmodell und Layout

Unter Verwendung des Controllers erstellt die Algoria Engine eine Instanz der als Plug-In geladenen Datenstruktur. Das Interface des Controllers ist in Listing 4 dargestellt. Die ersten drei *Properties* sind rein informativer Natur und werden von der Engine zur Auflistung der vorhandenen Datenstrukturen genutzt. Die *IsLinked* Eigenschaft definiert, ob ein Link notwendig ist, damit eine Datenstruktur mit einer anderen zusammenwachsen kann. Dies ist beispielsweise bei Listen und Bäumen der Fall, nicht jedoch bei Arrays. Das *Property SymbolsOfInterest* beinhaltet eine Menge von Symbolnamen, welche die Datenstruktur verarbeiten kann. Die Symbolnamen selber beziehen sich auf die in LADDER definierten Symbole.

Sobald die Engine ein Symbol erkannt hat, ruft sie die *Create*-Methode mit dem entsprechenden Symbol als Argument auf. Die *Create*-Methode generiert daraus eine neue Datenstruktur bestehend aus genau einem Symbol. Anschliessend überprüft die Engine mit der *TryMerge*-Methode, ob diese neu erzeugte Datenstruktur mit einer bereits bestehenden zusammenwachsen kann. Welche der bereits bestehenden Datenstrukturen für ein allfälliges Zusammenwachsen ausgewählt werden, hängt von einer Analyse der nächsten Nachbarn ab. Üblicherweise kommen nur Datenstrukturen und Links in Frage, die sich entweder berühren oder sehr nahe beinander liegen.

Die Modellierung einer Datenstruktur obliegt dem Plug-In-Entwickler. Es empfiehlt sich jedoch, das Modell so zu entwickeln, dass das GUI darauf binden und die Algorithmen wie gewohnt darauf operieren können. Das Algoria Framework stellt lediglich einige primitive Datentypen zur Verfügung, die bei der Modellierung einer Datenstruktur Verwendung finden. Zu diesen primitiven Datentypen gehören unter anderen *AnimNumber* und *AnimText*, die beide von *DependencyObject* erben und automatisch Ereignisse auslösen, die vom *AnimationProcessor* für die Animation verwendet werden können.

Das Layouten einer Datenstruktur ist eine nicht triviale Angelegenheit. Einerseits darf das gezeichnete Element nach dem Zeichnen nicht gleich davon springen (es könnte durch den Layout-Mechanismus an eine andere Position gezwungen werden), andererseits muss die Datenstruktur in der Lage sein, die Elemente sinnvoll auszurichten und verschönert darzustellen. Diesem Umstand Rechnung tragend unterscheidet das Layout-Panel typischerweise zwischen drei Modi: Position und Grösse unverändert übernehmen (Standard), Veränderungen an einem einzelnen Element und drittens Veränderungen an der ganzen Datenstruktur. Der letzte Modus wird beispielsweise dann benötigt, wenn die Grösse der gesamten visualisierten Datenstruktur interaktiv verändert wird. Wiederum stellt das Algoria Framework einige *AttachedProperties* zur Verfügung, die von den Plug-In-Entwicklern verwendet werden sollen.

Zum jetzigen Zeitpunkt ist die Entwicklung und Integration einer neuen Datenstruktur in Algoria noch mit sehr viel Aufwand verbunden. Erst mit der Entwicklung von weiteren Datenstrukturen wird sich zeigen, in welchem Masse sich gemeinsame Teile separieren und abstrahieren lassen, um fremden Datenstrukturentwicklern die Arbeit so weit wie möglich zu erleichtern. Im Gegensatz dazu ist die Entwicklung neuer Algorithmen, die auf den bestehenden Datenstrukturen operieren, relativ einfach und unterscheidet sich kaum, von der herkömmlichen Implementierung in anderen Software-Projekten.

Zusammenfassung und Ausblick

In der aktuellen Version von Algoria können Arrays und Listen von Hand skizziert, erkannt, mittels stiftfreundlichem Menü oder Gesten editiert und auf einfache Art animiert werden. Die zu erkennenden Symbole sind von Algoria separiert und in einer speziellen Symbolbeschreibungssprache (LADDER) abgelegt. Weitere Datenstrukturen wie Bäume und Graphen werden folgen. Alle Datenstrukturimplementierungen inklusive Algorithmen sind vom Algoria-Kern in separate Plug-

Ins ausgelagert, um individuelle Erweiterungen zu ermöglichen.

Vorerst ist es noch notwendig, Algoria die zu erkennende Datenstruktur mitzuteilen. Entsprechende Vorarbeiten zur Eliminierung dieser Zustandsabhängigkeit sind jedoch getroffen worden und sollen in den kommenden Versionen weiter umgesetzt werden.

Bei der Animation der Algorithmen verfolgen wir einen minimalistischen Ansatz, welcher ohne grossen Programmieraufwand auskommt. Der grösste Teil der Animation wird aus der herkömmlichen Formulierung der Algorithmen in C# automatisch generiert.

Referenzen

[AC10] Ambrósio, A.P.L., Costa, F. M. Evaluating the Impact of PBL and Tablet PCs in an Algorithms and Computer Programming Course. Proc. of the 41st ACM technical symposium on computer science education, SIGCSE, 2010.

[AU10] Ando, M., Ueno, M. Analysis of the Advantages of Using Tablet PC in e-Learning. 10th IEEE International Conference on Advanced Learning Technologie, ICALT, 2010.

[BBC10] Benlloch-Dualde, J.-V., Buendia, F., Cano, J.-C. Supporting instructors in designing Tablet PC-based courses. 10th IEEE International Conference on Advanced Learning Technologie, ICALT, 2010.

[COP09] Casas, I. Ochoa, S.F., Puente, J. Using Tablet PCs and Pen-Based Technologies to Support Engineering Education. Human-Computer Interaction. 13th Inter. Conf., 2009.

[GW00] Guimbretière, F., Winograd, T. FlowMenu: Combining Command, Text, and Data Entry. Proc. of the 13th Annual ACM Symposium on User Interface Software and Technology, 2000.

[Ham07] Hammond, T. Enabling Instructors to Develop Sketch Recognition Applications for the Classroom. Frontiers in Engineering, 2007.

[Ham07b] Hammond, T. LADDER: A Perceptually-Based Language to Simplify Sketch Recognition User Interfaces Development. MIT PhD Thesis, 2007.

[HD03] Hammond, T., Davis, R. LADDER. A Language to Describe Drawing, Display, and Editing in Sketch Recognition. Int. Joint Conf. on Artificial Intelligence, 2003.

[Hop91] Hopkins, D. The Design and Implementation of Pie Menus. Dr. Dobb's Journal, Dec. 1991.

[KB94] Kurtenbach, G., Buxton, W. User Learning and Performance with Marking Menus. Proc. of CHI '94, 1994.

[KFE07] Kurtz, B.L., Fenwick, J.B., Ellsworth, C.C. Using Podcasts and Tablet PCs in Computer Science. Proc. of the 45th Annual Southeast Regional Conference, 2007.

[KHPC08] Kamin, S., Hines, M., Peiper, C., Capitanu, B. A System for Developing Tablet PC Applications for Education. Proc. of the 39th SIGCSE Technical Symp. on Computer Science Education, 2008.

[Muk] Mukundan, R. Java Applets Centre. University of Canterbury, Computer Science. http://www.cosc.canterbury.ac.nz/mukundan/dsal/appldsal.html

[PH08] Paulson, B., Hammond, T. PaleoSketch: Accurate Primitive Sketch Recognition and Beautification. Proc. of the 2008 Int. Conf. on Intelligent User Interfaces, 2008.

[Rbe06] Rbeiz, M.A. Semantic Representation of Digital Ink in the Classroom Learning Partner. MIT MSc Thesis, Dept. of Electrical Engineering and Computer Science, 2006.

[Sch10] Schweizer, R. Algoria – Skizzenerkennung für Tablet-PCs. Symbolerkennung. Master Projekt P7a, 2010. http://webapache.imvs.technik.fhnw.ch/~christoph.stamm/reports/P7a_2010_Algoria.pdf