Ein System zur Visualisierung von Algorithmen und Datenstrukturen mit LEDA

Stefan Näher[1] and Oliver Zlotowski[2]

[1] Fachbereich IV-Informatik
Universität Trier
D-54286 Trier

[2] Martin-Luther-Universität Halle-Wittenberg
Fachbereich Mathematik und Informatik
Institut für Informatik
D-06099 Halle(Saale)

Zusammenfassung Diese Arbeit beschreibt einige grundlegende Konzepte und Implementierungsdetails des LEDA SV-Systems zur Visualisierung von Datenstrukturen und Algorithmen. Die wichtigsten Ziele dieses Projektes sind zum einen die Bereitstellung eines Systems zur Visualisierung von Verfahren, die für den Einsatz in der Lehre geeignet sind und zum anderen die Schaffung einer Plattform zur Unterstützung von Experimenten im Bereich des Algorithms Engineering. Zur Visualisierung werden neben der plattformunabhängigen Fensterklasse auch die Visualisierungswerkzeuge GraphWin[1] und GeoWin[2] von LEDA verwendet, so dass zusammen mit der Vielzahl von Algorithmen und Datenstrukturen in LEDA eine gute Ausgangsposition für die Entwicklung eines SV-Systems gegeben ist.

1 Einleitung

Die Leistung moderner Computersysteme hat die Entwicklung rechnergestützter Darstellungen komplizierter Sachverhalte in den letzten Jahren stark beschleunigt. So sind viele Bereiche der Industrie und Wirtschaft in hohem Maße von rechnergestützten Visualisierungen komplexer Sachverhalte abhängig, nicht zuletzt deshalb, weil „Computersysteme [...] Informationen als abstrakte Symbole [repräsentieren], [wobei] ihre Repräsentationsformen [...] in der Regel darauf ausgelegt [sind], Berechnungen und Analyseoperationen möglich zu machen und den Informationszugriff zu erleichtern.“ [RS99]

Die Visualisierung von Software ist ein Vertreter rechnergestützter Visualisierungen, wobei in den folgenden Betrachtungen ein Teilgebiet der Software Visualisierung (kurz: SV) im Vordergrund steht: die Animationen.

In der Lehre werden Animationen mit dem Ziel eingesetzt, das Verständnis der Funktionsweise von Algorithmen und Datenstrukturen zu fördern. Aus

[1] siehe Kapitel 2 in [MN99]
[2] siehe [Bae]

pädagogischer Sicht gehört zu den wichtigsten Vorteilen rechnergestützter Darstellungen die Möglichkeit des Einsatzes von bewegten Grafiken, die durch den Benutzer interaktiv gesteuert werden können [MH94]. Hierbei lassen sich die Vorteile eines Buches (Vor-/Zurückblättern und Bestimmung der persönlichen Lerngeschwindigkeit) mit denen eines Films (bewegte Grafiken, Töne und Spezialeffekte) auf sinnvolle und kreative Art verbinden.

Neben der Kreativität des Entwicklers von Animationen hat aber auch die Beachtung mediendidaktischer und psychologischer Aspekte bei der Erstellung einen entscheidenden Einfluss auf die Qualität und den Nutzen einer Animation. Aus dem Bereich der Pädagogik und der Informatik gibt es daher zahlreiche Erfahrungen mit dem Einsatz von Animationen in der Lehre. Der interessierte Leser sei hier beispielsweise auf [BCS96,LBS94,SDBP98,KST99] verwiesen.

Der nächste Abschnitt gibt auf Basis der in [PBS93] vorgestellten Taxonomie einen kurzen Überblick über die Teilbereiche der SV. Im Anschluss daran werden einige im Rahmen einer laufenden Diplomarbeit erstellten Grundkomponenten für die Entwicklung von Animationen unter Verwendung von LEDA[3] vorgestellt und an einem Beispiel ihre Verwendung demonstriert.

2 Teilgebiete der Software Visualisierung

Das Ziel der Software Visualisierung besteht im allgemeinen darin, den Verstehensprozess für die Arbeitsweise von Software zu erleichtern. Um dieses Ziel zu erreichen, werden häufig Techniken aus der Typographie und der Kinematographie angewandt und mit modernen Methoden der Mensch-Computer-Interaktion gekoppelt [SDBP98].

Die SV lässt sich, wie bereits erwähnt, in verschiedene Teilbereiche gliedern, die in der folgenden Abbildung dargestellt sind:

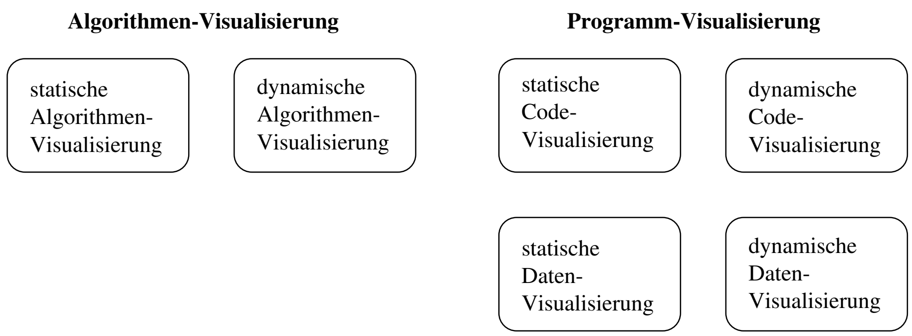


Abbildung1. Die Teilgebiete der Software Visualisierung [PBS93].

[3] Library of Efficient Data types and Algorithms [MN99]

Die Visualisierung von Algorithmen beinhaltet hierbei die Beschreibung von Software auf einem hohen Abstraktionsgrad, d.h. dass im wesentlichen nur strukturelle Ähnlichkeiten zwischen den Darstellungselementen und dem implementierten Verfahren bei Vertretern dieses Visualisierungstyps zu finden sind. Das Teilgebiet lässt sich weiter in statische- und dynamische Algorithmen-Visualisierung untergliedern, wobei letzterer Visualisierungstyp auch als Algorithmen-Animation bezeichnet wird. Das Video *Sorting Out Sorting* von Baecker vorgestellt in [Bae81] gilt als eines der ersten Vertreter von Algorithmen-Animationen. In diesem Video wird die Arbeitsweise verschiedener Sortierverfahren mit Hilfe einfacher animierter Darstellungen erläutert.

Ein einfaches Beispiel für eine statische Algorithmen-Visualisierung ist ein Programmablaufplan. Im Sinne des Begriffes Abstraktion werden in der Regel nur wesentliche Verfahrensschritte in statischer Form unter Verwendung einer grafischen Symbolik dargestellt. Jeder Bezug auf eine konkrete Programmiersprache oder Implementierungsvariante wird dadurch vermieden. In Lehrbüchern findet dieser Visualisierungstyp oft Anwendung.

Auf der rechten Seite in der Abbildung 1 ist das zweite Teilgebiet der SV dargestellt, die Programm-Visualisierung. Der Unterschied zwischen den beiden Visualisierungstypen soll durch eine Definition des Algorithmenbegriffes verdeutlicht werden: Algorithmen sind exakt formulierte Rechenvorschriften, die auf Computern als ausführbare Programme implementiert und auf diese Art für die Lösung von Aufgaben angewandt werden können. „Ein Algorithmus besteht somit aus einer wohldefinierten endlichen Folge von elementaren Rechenoperationen und Entscheidungen, um aus einer bestimmten Menge von Eingabegrößen [...] das gewünschte Resultat [...] zu erzeugen.“ [Zei96]

Programme sind hiernach nichts anderes als Implementierungsvarianten von Algorithmen. Im Vordergrund einer Programm-Visualisierung steht somit die Darstellung einer konkreten Implementierung eines Algorithmus. Das wesentliche Unterscheidungsmerkmal zwischen beiden Visualisierungstypen ist deshalb der Abstraktionsgrad, der bei einer Programm-Visualisierung in der Regel geringer ist als bei einer Algorithmen-Visualisierung. Die Untergliederung des Teilgebietes in Code- und Daten-Visualisierung weist zusätzlich auf die Techniken der Programm-Visualisierung hin. Die Einteilung erscheint sinnvoll, da zur Verwaltung und Berechnung von Daten oft charakteristische Datenstrukturen im Programm verwendet werden, deren Arbeitsweise in gesonderten Darstellungen die Vermittlung der Grundidee einer konkreten Implementierung unterstützen können. Daten-Visualisierungen sind sehr vielfältig. So kann beispielsweise eine Liste durch Rechtecke dargestellt werden, wobei die Verkettung der Listenelemente intuitiv durch Pfeile erfolgt. Werden zwei Elemente in einem Verfahrensschritt vertauscht, animiert die dynamische Daten-Visualisierung diesen Austausch durch einen graduellen Vorgang.

Was ist nun eine Code-Visualisierung? Ein einfaches Beispiel für eine statische Code-Visualisierung ist Darstellung des Programmtextes. Das kontinuierliche Hervorheben des gerade aktiven Programmteils kann als Code-Animation interpretiert werden. Auch die Darstellung eines Rekursionsbaumes bei einem rekursiven Verfahren kann als ein Vertreter dieses Visualisierungstyps aufgefasst werden.

Abschliessend sei noch daraufhingewiesen, dass in der Praxis bei vielen existierenden Animationen keine klare Zuordnung zu einem der Teilgebiete vorgenommen werden kann, da viele Visualisierungen sowohl stark abstrahierende, als auch sehr an der Implementierung orientierte Darstellungselemente besitzen.

3 Komponenten des LEDA SV-Systems

In diesem Abschnitt werden einige der umgesetzten Konzepte anhand bereits implementierter Komponenten erläutert. Doch zuvor wird die Frage beantwortet, warum LEDA als Plattform für die Entwicklung eines SV-Systems geeignet erscheint.

Aus der Sicht eines Software-Entwicklers ist LEDA eine Plattform für die Lösung kombinatorischer und geometrischer Probleme. Durch eine einfache Transformation ist der Entwickler in der Lage, Algorithmen in effiziente C++-Programme zu überführen, da ihm hierfür eine Vielzahl von einfachen (Stack, Liste) und komplexen (Graph, Wörterbuch) Datenstrukturen zur Verfügung stehen. Diese Datenstrukturen sind als unabhängige Datentypen in Form von C++-Klassen implementiert. Dadurch hat der Entwickler die Möglichkeit über die Schnittstelle der Klasse einer Datenstruktur charakteristische Operationen auf ihr auszuführen.

Eine Erweiterung der in sich gekapselten Strukturen um neue Funktionalität ist durch das Vererbungskonzept von C++ möglich. Diese Programmiermethodik wird bei den folgenden Komponenten immer wieder angewendet, um den großen Umfang bestehender Datentypen in LEDA für Visualisierungen nutzbar zu machen.

Die einfache Programmierung der grafische Schnittstelle ist ein weiteres Merkmal von LEDA. Unter Verwendung der Fensterklasse hat der Entwickler die Möglichkeit verschiedene grafische Schnittstellensysteme (Windows 95/98/NT, Unix/Linux) zu nutzen.

Die große Menge an Datenstrukturen sowie die einfache Grafikprogrammierung sind zwei wesentliche Punkte, die den Versuch des Designs eines SV-Systems auf der Basis von LEDA sinnvoll erscheinen lassen. Für einen tieferen Einblick in das LEDA System sei der interessierte Leser auf [MN99,MNSU99] verwiesen.

Bevor im folgenden einige konkrete Konzepte erläutert werden, formuliert der nächste Abschnitt die vorläufigen Entwicklungsziele für das System.

3.1 Entwicklungsziele

Viele untersuchte Systeme wie Balsa [Bro88], Zeus [Bro91], Tango [Sta92,Sta89] oder Polka [Sta93], sowie die in [PBS93] vorgestellte Taxonomie beeinflussten die Herausarbeitung der folgenden Entwicklungsziele:

▹ *Wiederverwendbare und unabhängige Komponenten*
Die implementierten Komponenten sollten für verschiedene Algorithmen einer

Algorithmenklasse wiederverwendbar sein und in einer dokumentierten Bibliothek zusammengefasst werden. Dies betrifft insbesondere die *Views* sowie ihre Steuerung und Verwaltung.

▷ *Einfache Erweiterung bzw. Anpassung der Visualisierungskomponenten*
Die Konzepte sollten so umgesetzt werden, dass neue Funktionalität (z.B. eine neue Darstellung) dem System hinzugefügt werden kann, ohne dass bestehende Komponenten modifiziert werden müssen.

▷ *Interaktionsschnittstellen*
Über eine Interaktionsschnittstelle sollte es dem Anwender möglich sein, die Animation zu steuern, d.h. vorerst im wesentlichen die Eingabedaten sowie die Geschwindigkeit und die Richtung zu kontrollieren.

▷ *Minimale Modifikationen am Verfahren*
Ein bereits unter Verwendung von LEDA Datenstrukturen implementiertes Verfahren sollte durch minimale Veränderungen in eine einfache Animation überführbar sein.

Das letzte Entwicklungsziel lässt sich nur bis zu einem gewissen Grad erreichen, denn komplexe Animationen, die insbesondere für den Einsatz in der Lehre konzipiert werden, benötigen zusätzliche Informationen, die vom Verfahren selbst nicht erzeugt werden. Diese abundanten Attribute einer Animation müssen somit zusätzlich in den Programmtext eingefügt werden. Daher erscheint es sinnvoll, auch für diese Art von Information, leicht verwendbare Visualisierungskomponenten zur Verfügung zu stellen. Zum Abschluss der Betrachtungen wird auf diese Problematik noch einmal kurz eingegangen.

3.2 Visualisierung von Datenstrukturen

Der wesentliche Vorteil bei der Visualisierung von Datenstrukturen besteht darin, dass ein Großteil der zur Visualisierung benötigten Informationen aus dem Zustand der Struktur gewonnen werden kann. Hierbei informiert die Datenstruktur, im folgenden auch als *Client* bezeichnet, durch das Auslösen verschiedener *Events* spezielle Komponenten über eine Zustandsveränderung. Diese Komponenten werden *Observer* genannt, deren Aufgabe es ist, die mit dem jeweiligen *Event* transportierten Informationen zu verarbeiten und eventuell an geeignete Visualisierungskomponenten (*Views*) oder an andere *Observer* weiterzuleiten.

Die Kommunikation zwischen dem *Client* und dem *Observer* ist ein typisches Anwendungsbeispiel für das Beobachtungsmuster (engl. *Observer Pattern*) aus [GHJV95]. Eine allgemeine Implementierungsvariante dieses Musters war unter anderem Gegenstand der Diplomarbeit von A. Luleich. Mit der Implementierung des Datentyps `EVENTx` stellt LEDA eine Möglichkeit des $1:n$ Informationsaustausches zwischen beliebigen Objekten zur Verfügung. Die Abbildung 2 zeigt das *Event*-System in der OMT-Notation[4].

An dieser Stelle wird auf die Funktionsweise des *Event*-Systems nicht näher eingegangen. Der interessierte Leser sei auf [Lul98] verwiesen. Statt dessen wird

[4] Object Modeling Technique

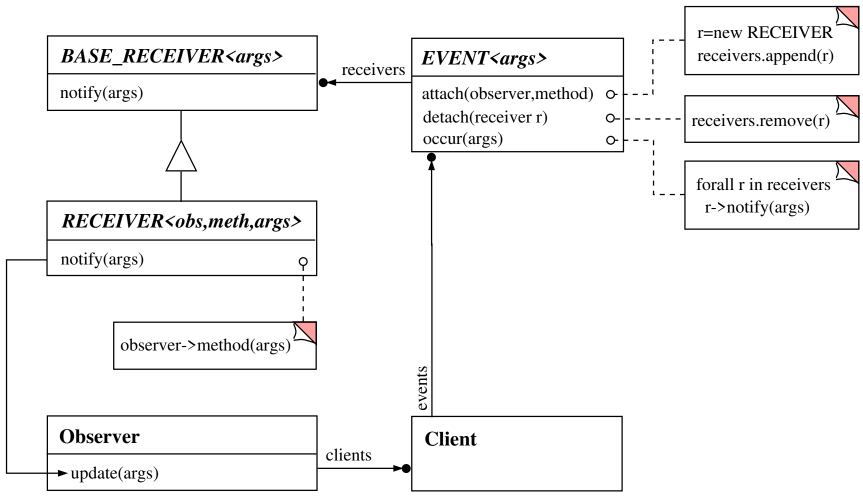


Abbildung2. Das Klassendiagramm des *Event*-Systems in LEDA [Lul98].

unter Benutzung dieses Systems und der LEDA Datenstruktur `array<E>` die Klasse `event_array<E>` betrachtet.

3.3 Event Array

Die Animation von Sortierverfahren ist ein beliebtes Einsatzgebiet eines SV-Systems, da dieser Typ von Algorithmus in der Regel auf Feldern arbeitet und sich Felder einfach visualisieren lassen. Aus diesem Grund dient der parametrierte LEDA Datentyp `array<E>` als Basis für die Template-Klasse `event_array<E>`, der ersten Grundkomponente des Visualisierungssystems. Der Template-Parameter `E` bestimmt im folgenden immer den vom Feld verwalteten Datentyp.

Durch Ableitung von der Klasse `array<E>` erbt die Klasse `event_array<E>` die Grundfunktionalität des Basistyps. Alle Member-Funktionen der Klasse `event_array<E>` (mit Argumenten) wurden zusätzlich um *Events* erweitert, so dass bei jedem Zugriff auf die Datenstruktur die zugehörigen *Events* ausgelöst werden.

Die Menge der definierten *Events* wird als *Event*-Kollektion bezeichnet, in der grundsätzlich zwei Typen zu unterscheiden sind:

▷ *Events zur Verwaltung*
Diesem Typ von *Event* gehört der `construct`-, `destruct`- und der `update`-*Event* an. Sie dienen einem *Observer* zur Verwaltung seiner *Clients*. So kann sich ein *Client* an einen *Observer* an- bzw. abmelden. Das `update`-*Event* wird von der Funktion `enable_events()` ausgelöst und dient dazu, den *Observer* über eventuell komplexe Veränderungen in der Datenstruktur zu informieren, die ihm durch einen Aufruf der Funktion `disable_events()` entgangen sind.

▷ *Events in Member-Funktionen für den Zugriff auf das Feld*
Der zweite Typ von *Event* wird in Member-Funktionen ausgelöst, die entweder die Datenstruktur direkt verändern oder Informationen über einzelne Feldelemente zurückliefern. So löst zum Beispiel jede Vergleichsoperation einen `compare`-*Event* aus.

Durch einen `prev_Member_Funktions`-*Event* kann nun ein *Observer* beispielsweise die Gültigkeit der übergebenen Argumente überprüfen und bei ungültigen Argumenten (zumeist Bereichsüberschreitungen) das `_error`-Flag setzen, so dass nach der Rückkehr aus der *Event*-Behandlung die Basisfunktion nicht ausgeführt wird (vgl. Abbildung 3).

Eine zweite Funktion dieser *Events* besteht darin, einem *Observer* die Möglichkeit zu geben, den Zustand der Datenstruktur vor einer Veränderung zu speichern. Diese *Undo*-Information ist für eine Interaktionskomponente wertvoll, da hiermit Veränderungen in der Datenstruktur auch rückgängig gemacht werden können. Nachdem die Basisfunktion ausgeführt wurde, hat der *Observer* die Möglichkeit den aktuellen Zustand zum Zwecke einer *Redo*-Operation zu speichern, hierzu löst die Funktion den `post_Member_Funktions`-*Event* aus. Natürlich benötigen nicht alle Funktionen (z.B. `swap()`) einen `prev_`- und einen `post_Member_Funktions`-*Event*, um die beschriebenen Anwendungsmöglichkeiten zu gewährleisten.

Am Beispiel zweier Funktionen wird der Einsatz dieser *Events* gezeigt:

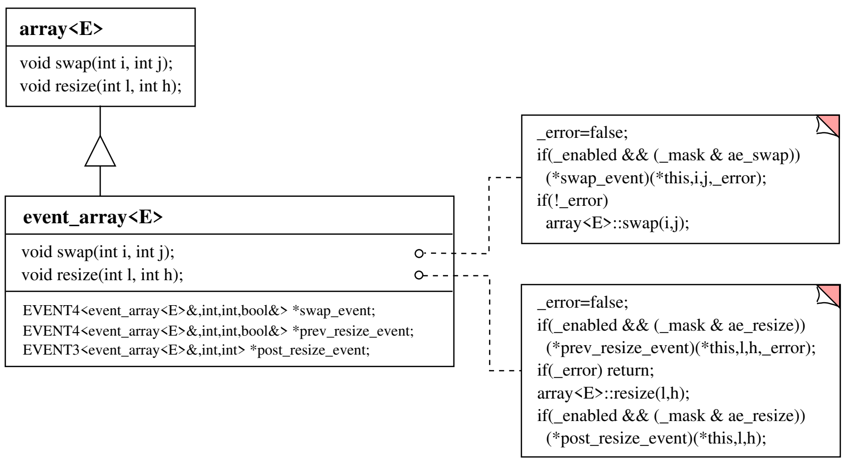


Abbildung3. Ein Klassendiagramm-Ausschnitt des Datentyps `event_array<E>`.

Nachdem nun ein *Client* betrachtet wurde, stellt der nächste Abschnitt einen passenden *Observer* vor.

3.4 Array Observer und Array View

Die Klasse `array_observer<E,View>` ist ein *Observer*, der nun auf die ausgelösten *Events* eines *Client*s reagiert und die mit dem jeweiligen *Event* transportierten Informationen an spezielle Visualisierungskomponenten weiterleitet. In Abbildung 4 sind die Beziehungen zwischen den einzelnen Klassen dargestellt:

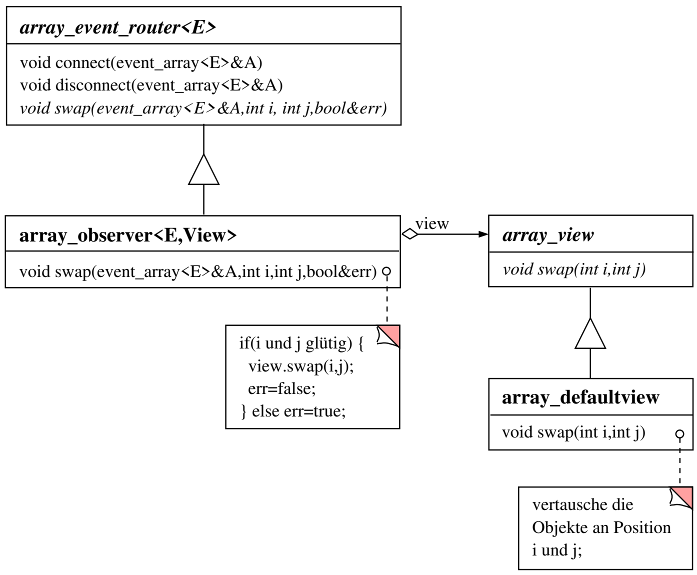


Abbildung4. Die Beziehungen zwischen den einzelnen Klassen.

Die Klasse `array_observer<E>` ist eine Unterklasse eines *Eventrouter*s, dessen Funktionalität hier nicht näher betrachtet wird, da sich dessen Aufgabe im wesentlichen auf den Verbindungsaufbau `connect()` bzw. Verbindungsabbau `disconnect()` zu einzelnen *Clients* beschränkt.

Durch das Überschreiben der virtuellen *Event*-Funktionen der Basisklasse hat der *Observer* nun die Möglichkeit, gezielt Informationen über den Zugriff auf die Datenstruktur an die Visualisierungskomponente weiterzuleiten. In Abbildung 4 wird dieses Vorgehen am Beispiel der *Event*-Behandlung für den `swap`-*Event* gezeigt.

Die Klasse `array_view` stellt eine Basisklasse für verschiedene *Views* dar, so dass durch Schnittstellenvererbung die Klasse `array_defaultview` Informationen über einen *Client* empfangen und diese in einer grafischen Standard-Darstellung für kleine Felder verarbeiten kann.

Der folgende Programmausschnitt demonstriert die Anwendung der Klassen:

```
event_array<int> A(10);
array_defaultview View;
array_observer<int,array_defaultview> Observer(View);
Observer.connect(A);

// Initialisierung des Feldes und Öffnen des Views

A.swap(3,9);
```

Das vollständige Programm liefert die in Abbildung 5 dargestellte Momentaufnahme während des Austausches:

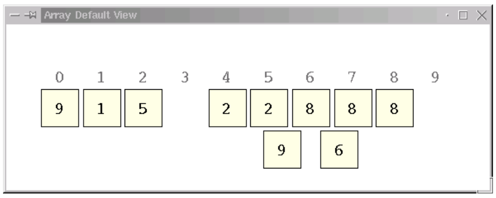


Abbildung5. Eine Momentaufnahme während eines Austausches.

Durch die Bereitstellung der Klasse `array_observer<E,View>` und `array_view` hat der Entwickler nun die Möglichkeit, spezielle Darstellungen für Felder zu entwickeln, wobei ihm dabei viele Informationen aus der Datenstruktur automatisch zur Verfügung gestellt werden, ohne dass dazu umfangreiche Anpassungen im Programmtext vorgenommen werden müssen.

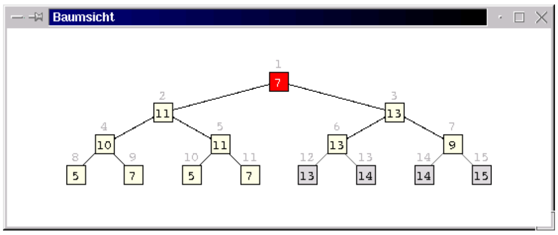
Als Unterstützung für die Entwicklung eigener *Views* stehen dem Entwickler neben der Fensterklasse auch die Visualisierungswerkzeuge GraphWin und GeoWin zur Verfügung, die durch eine breite Funktionalität das Modellieren und Designen eines *View*s vereinfachen.

4 Anwendungsbeispiel: Heapsort

Bei der Animation des Sortierverfahrens Heapsort bietet sich neben der im letzten Abschnitt vorgestellten einfachen Feldsicht, auch die Sicht des Feldes aus der Baumperspektive an. Möglicherweise ist diese Sicht des Feldes auch sinnvoller, da Vertauschungen in der einfachen Feldsicht willkürlich wirken, in der Baumsicht jedoch die Idee des Verfahrens besser veranschaulicht wird.

Der folgende Programmtext erzeugt unter Verwendung eines weiteren *View*s (implementiert durch die Klasse `array_bintreeview`) und des bereits beschriebenen *View*s die dargestellte Momentaufnahme in Abbildung 6.

```
#include <LAAP/array_views.h>

event_array<int> A(1,15);
array_bintreeview TreeView("Baumsicht");
array_defaultview DefaultView("Feldsicht");
array_observer<int,array_bintreeview> Observer1(A,TreeView);
array_observer<int,array_defaultview> Observer2(A,DefaultView);
ao_client obs_client1,obs_client2;

void set_item_color(int i,color c=ivory)
{ TreeView.item_bg_color(obs_client1,i,c);
  DefaultView.item_bg_color(obs_client2,i,c);
}

void heapify(int k,int N)
{ set_item_color(k,red);      // Vater ist rot markiert
  while(2*k<=N) {
    int j=2*k;
    if(j<N && A.smaller(j,j+1)) j++;
    if(A.geq(k,j)) break;
    A.swap(k,j);
    k=j;
  }
  set_item_color(k);          // Vater wieder unmarkiert
}

int main()
{ obs_client1=Observer1.get_client(A);
  obs_client2=Observer2.get_client(A);

  // Initialisierung des Feldes und Öffnen der Views

  int N=A.high();
  for(int i=N/2;i>=A.low();i--)
    heapify(i,N);
  while(N>1) {
    A.swap(N,1);
    set_item_color(N,grey1);  // graue Feldelemente (sortiert)
    heapify(1,--N);
  }
  // ...
}
```



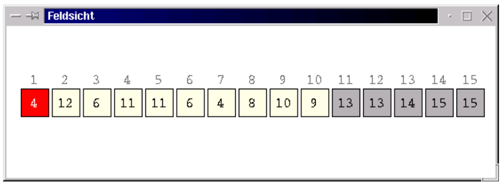


Abbildung6. Oben die Ausgabe von `TreeView` und unten die von `DefaultView`.

5 Zusammenfassung und Ausblick

Anhand des Beispiels wurde demonstriert, dass durch minimale Veränderungen im Programmtext eine einfache Algorithmen-Animation erstellt werden kann. Der Betrachter der Animation hat aber bislang noch keine Möglichkeit, mit der Animation zu interagieren. Für eine sinnvolle Bereitstellung einer solchen Interaktionsschnittstelle sind vielleicht weitere Veränderungen im Programmtext notwendig. Gegenwärtig werden für dieses Problem bereits vorgeschlagene Konzepte auf ihre Anwendbarkeit hin überprüft, so dass in naher Zukunft erste Fortschritte in diesem Bereich zu erwarten sind.

Durch das Bestreben beim Design des Systems möglichst allgemein verwendbare Konzepte zu finden, ist eine Übertragung des Vorgehens auf andere Datenstrukturen und damit andere Algorithmenklassen möglich. Ein Beispiel hierfür ist die LEDA Datenstruktur `graph`, in der ebenfalls eine *Event*-Kollektion definiert ist, so dass der zugehörige GraphWin-*Observer* die Veränderungen eines Graphen mit Hilfe des Visualisierungswerkzeuges GraphWin veranschaulichen kann.

Das allgemeine Ziel des Projektes ist die Bereitstellung eines modernen Systems zur Visualisierung von Algorithmen und Datenstrukturen, um es in der Lehre und bei Experimenten im Bereich des Algorithms Engineering einsetzen zu können.

Literatur

[Bae] M. Baesken. Interaktive Visualisierung geometrischer Algorithmen mit GeoWin. in Vorbereitung.

[Bae81] R. Baecker. Sorting Out Sorting, 1981.

[BCS96] M. Byrne, R. Catrambone, and J. Stasko. Do Algorithm Animations Aid Learning? Technical Report GIT-GVU-96-18, GVU Center, Georgia Institute of Technology, Atlanta, GA, August 1996.

[Bro88] M. Brown. *Algorithm Animation.* MIT Press, New York, 1988.

[Bro91] M. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 4–9, Kobe, Japan, October 1991.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 1995.

[KST99] C. Kehoe, J. Stasko, and A. Taylor. Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study. Technical Report GIT-GVU-99-10, GVU Center, Georgia Institute of Technology, Atlanta, März 1999.

[LBS94] A. Lawrence, A. Badre, and J. Stasko. Empirically Evaluating the Use of Animations to Teach Algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages 48–54, St. Louis, MO, October 1994.

[Lul98] A. Luleich. Visualisierung von Graphalgorithmen. Master's thesis, Martin-Luther-Universität Halle-Wittenberg, Halle(Saale), Dezember 1998.

[MH94] A. Mester and P. Herrmann. Audiovisuelle Animation von verteilten Algorithmen und Kommunikationsprotokollen. Technical Report Projektgruppe 225, Fachbereich Informatik, Universität Dortmund, Dortmund, April 1994.

[MN99] K. Mehlhorn and S. Näher. *LEDA: A Plattform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[MNSU99] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual Version 4.0.* Saarbrücken, 1999.

[PBS93] B. Price, R. Baecker, and I. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, April 1993.

[RS99] K. Reichenberger and R. Steinmetz. Visualisierungen und ihre Rolle in Multimedia–Anwendungen. *Informatik Spektrum*, pages 88–97, April 1999.

[SDBP98] J. Stasko, J. Domingue, M. Brown, and B. Price. *Software Visualization.* MIT Press, Cambridge, London, 1998.

[Sta89] J. Stasko. *TANGO: A Framework and System for Algorithm Animations.* PhD thesis, Brown University,Providence, RI, May 1989. Available as Technical Report No. CS-89-30.

[Sta90] J. Stasko. The Path Transitions Paradigm: A Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, September 1990.

[Sta92] J. Stasko. Animating Algorithms with XTANGO. *SIGACT News*, 23(2):67–71, Spring 1992.

[Sta93] J. Stasko. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[Zei96] E. Zeidler. *Teubner-Taschenbuch der Mathematik.* B. G. Teubner, Stuttgart, Leipzig, 1996.

[Zlo99] O. Zlotowski. Animation der Sortierverfahren Heapsort und Quicksort unter Verwendung von LEDA. In *Informatikertage 1999, Fachwissenschaftlicher Informatik-Kongress*, pages 192–194, Bad Schussenried, März 1999.