Inhaltsverzeichnis

1 Hintergründe und Motivation für Kompressionsverfahren

Die Hauptnotwendigkeit für die Entwicklung von Kompressionsverfahren diskreter bzw. kontinuierlicher Medien ist in der Anpassung des Datenvolumens einer allgemeinen Datenübertragung an die freien Kapazitäten des Netzes bzw. der Endgeräte zu sehen. Das komprimierte Medium liegt bei seiner Übertragung im Zustand eines deutlich geringeren Datenvolumens vor als zum Zeitpunkt nach der Dekompression beim Empfänger. Dadurch ist eine geringere Netzbelastung gewährleistet und die Übertragungszeit nimmt ab.

Die Kompressionsidee liegt zusammenfassend darin, das zu übertragende Medium auf seinem Weg durch das Netz in datenvolumenverringerte Teilinformationen zu transformieren. Durch geeignete Dekompressionsalgorithmen hat der Empfänger die Möglichkeit, das Medium zu rekonstruieren, wenn auch u. U. auf Kosten seiner ursprünglichen Qualität. Man unterscheidet zwischen verlustfreien Kompressionsverfahren, durch die das Original perfekt wiederhergestellt werden kann, und verlustbehafteten Kompressionsverfahren. In diesem Fall existiert ein Unterschied zwischen Originalobjekt und dekodiertem Objekt. Zusätzlich wird dabei ausgenutzt, daß der Empfänger mit den Möglichkeiten seiner sinnlichen Wahrnehmung diese Diskrepanz unter Umständen gar nicht mehr ausmachen kann. Dies hängt allerdings von der situativen Spezifikation der Kompression ab.

2 Übersicht und Erläuterung der verwendeten Kompressionsalgorithmen für Standbilder

2.1 Block Truncation Coding (BTC)

Dieses Verfahren wird auf Graustufen-Bilder angewendet. Jedes Pixel sei dabei im Original mit einem Graustufenwert von 0 (schwarz) bis 255 (weiß) beschrieben. Das Bild wird in Blöcke der Größe $n \times m$ Pixel unterteilt. Folgendes Vorgehen wird auf jeden dieser Blöcke angewendet. Mittelwert und Standardabweichung der Pixelwerte für jeden Block werden nach folgenden Formeln berechnet:

$$\mu = \frac{1}{nm} \sum_{i=1}^{n} \sum_{j=1}^{m} Y_{i,j}$$

$$\sigma = \sqrt{\frac{1}{nm} \sum_{i=1}^{n} \sum_{j=1}^{m} Y^2{}_{i,j} - \mu^2}.$$

Mit $Y_{i,j} =$ Helligkeit (Graustufe) des Pixels. Danach wird dem jeweiligen Block eine Bitmatrix der Größe $n \times m$ nach folgender Regel zugewiesen:

$$B_{i,j} = \begin{cases} 1 \ldots \mathit{falls}\ Y_{i,j} \leq \mu \\ 0 \ldots \mathit{sonst.} \end{cases}$$

Nun werden lediglich zwei Graustufenwerte für den Block berechnet, a als Wert für seine dunkleren Pixel, b für seine helleren.

$$a = \mu - \sigma \sqrt{\frac{p}{q}}$$

$$b = \mu + \sigma \sqrt{\frac{q}{p}}$$

Hierbei repräsentiert p die Anzahl der Pixel, die heller als der Mittelwert μ sind, q entspricht der Anzahl der dunkleren Pixel. Als Resultat der Anwendung des Algorithmus auf jeden Block ergibt sich also die Bitmatrix und die blockspezifisch ermittelten beiden Graustufenwerte a und b.
Die Dekompression eines Blockes funktioniert nun nach folgendem Schema:

$$Y'_{i,j} = \begin{cases} a \ldots \mathit{falls} \quad B_{i,j} = 1 \\ b \ldots \mathit{sonst.} \end{cases}$$

Damit ist $Y'_{i,j}$ der dekodierte Graustufenwert.

Ein Beispiel verdeutlicht die Kompressionsrate von BTC:
Gegeben sei eine Blockgröße von 4×4 Pixeln. Das Original beansprucht demnach einen Speicherbedarf von 16 Bytes, da jedes Pixel mit einem Byte kodiert ist. In seiner kodierten Darstellung beansprucht dieser Block dagegen nur 16 Bits für die Bitmatrix, denn diese enthält nur die Elemente 1 und 0, und je ein Byte für die Graustufenwerte a und b. Also insgesamt vier Byte.

Trotz guter Kompressionsraten bei sehr groß gewählten Blöcken wirkt sich dies i.d.R. nachteilig auf die Rekonstruktion aus, da sich ein Block über beliebig inhomogene Bereiche des Bildes erstrecken kann und trotzdem nur zwei Graustufenwerte zu seiner Dekodierung zur Verfügung stehen.

2.2 Color Cell Compression (CCC)

Dieses Verfahren ist für die Kompression von Farbbildern vorgesehen. Hierbei ist jeder Pixelwert mit drei Byte gespeichert, jeweils ein Byte für die drei Farben Rot, Grün und Blau. Der Algorithmus CCC wird auf Blöcke der Größe $n \times m$ Pixel angewendet. Zunächst wird für jedes Farbpixel sein Helligkeitseindruck , d.h. der Erfahrungswert der subjektiven Wahrnehmung eines Menschen, berechnet:

$$Y = 0.3P_{red} + 0.59P_{green} + 0.11P_{blue}$$

Die mittleren Farbwerte der Pixel werden mit c = red, green, blue berechnet gemäß

$$a_c = \frac{1}{q} \sum_{Y_{i,j} \leq \mu} P_{c,i,j}$$

$$b_c = \frac{1}{p} \sum_{Y_{i,j} > \mu} P_{c,i,j} \ .$$

Dabei entspricht p der Anzahl der Pixel, die heller als der Mittelwert sind und q der Anzahl der dunkleren Pixel. Analog zum vorherigen Verfahren wird dem Block eine Bitmatrix der Größe $n \times m$ nach folgender Regel zugewiesen:

$$B_{i,j} = \begin{cases} 1 \ldots falls \quad Y_{i,j} \leq \mu \\ 0 \ldots sonst. \end{cases}$$

Die bereits ermittelten Werte $a = (a_{red}, a_{green}, a_{blue})$ und $b = (b_{red}, b_{green}, b_{blue})$ werden in eine Farbtabelle abgebildet, Color Lookup Table (CLUT). Von Belang sind nun die Werte a' und b', die die Einträge dieser Farbtabelle und damit die beiden ermittelten Farbwerte indizieren. Bei der Ausgabe dieses Verfahrens handelt es sich wieder um die Bitmatrix und die Indizes a' und b' für jeden Block.

Da es sich bei den Indizes natürlich nicht um Farbwerte, sondern nur um Platzanweiser handelt, genügt für sie ein Speicherbedarf von einem Byte, falls die Farbtabelle 256 Einträge enthält.

Beispiel: Man betrachte nun einen 4x4 Block, der insgesamt 16 Pixel beeinhaltet. Der Farbwert eines solchen Pixels ist also nach Definition im Original mit drei Byte abgespeichert. Der Block bedarf also einer Speicherkapazität von 48 Byte. Nach Kompression durch CCC ergibt sich für den Block ein wesentlich geringerer Speicherbedarf: Die Bitmatrix mit den Elementen 1 bzw. 0 benötigt lediglich 16 Bits. Die beiden Indizes, die auf die beiden berechneten mittleren Farbwerte für den Block in der Farbtabelle zeigen, benötigen ein Byte. Insgesamt also eine beachtliche Reduktion des Speicherplatzes für den Block von 48 Byte auf vier Byte.

Die Dekompression für jeden Block errechnet sich wie folgt:

$$P'_{i,j} = \begin{cases} CLUT[a'], falls & B_{i,j} = 1 \\ CLUT[b'], sonst. \end{cases}$$

Falls nun die Farbwerte der Pixel innerhalb eines Blockes im Original nur schwach voneinander abweichen, lohnt sich die Kodierung durch CCC mit der gewählten Blockgröße und die Dekompression für diesen Block mit nur zwei Farbwerten. Dennoch kann der Fall eintreten, daß die Farbwerte der Pixel in einem kleineren Bildbereich stark und häufig voneinander abweichen. Bei relativ groß gewählten Blöcken macht sich also nach der Dekompression ein deutlicher Qualitätsverlust bemerkbar, da die ursprüngliche Farbvielfalt durch zwei Farbwerte nicht annähernd erreicht werden kann.

Diese Überlegungen und Verbesserungsmöglichkeiten gaben Anlaß zur Entwicklung des folgenden Kompressionsverfahrens. Das bisher vorgestellte Verfahren CCC wird dabei adaptiv gemacht, paßt sich also an die speziellen lokalen Gegebenheiten des Bildes an.

2.3 Extended Color Cell Compression (XCCC)

Prinzip:
Größere Blöcke werden zunächst mit CCC kodiert. Falls die Abweichung der tatsächlichen Farbwerte im Block von a’ und b’ größer als ein vorgegebener Schwellenwert ist, der aus dem Produkt aus einem Wert zwischen zehn und zwanzig und den Blockgrößen besteht, wird dieser Block in vier Subblöcke zerlegt. Bezieht sich ein Block über einen homogenen Bildbereich, besteht keine Notwendigkeit, ihn weiter zu unterteilen. Erstreckt er sich allerdings über ein inhomogenes Bildgebiet, was seine Unterteilung in kleinere Blöcke wahrscheinlicher macht, ergibt sich für den ursprünglichen Block eine schlechtere Kompressionsrate, aber sein Dekompressionsergebnis nähert sich offensichtlich der Originalqualität stärker an. Die Kernidee von XCCC ist also die adaptive Blockgröße.
Analog zu den bereits vorgestellten Verfahren wird das Bild zunächst in größere Blöcke (i.d.R. 16×16 Pixel) unterteilt. Der Algorithmus wird auf jeden dieser Blöcke angewendet.

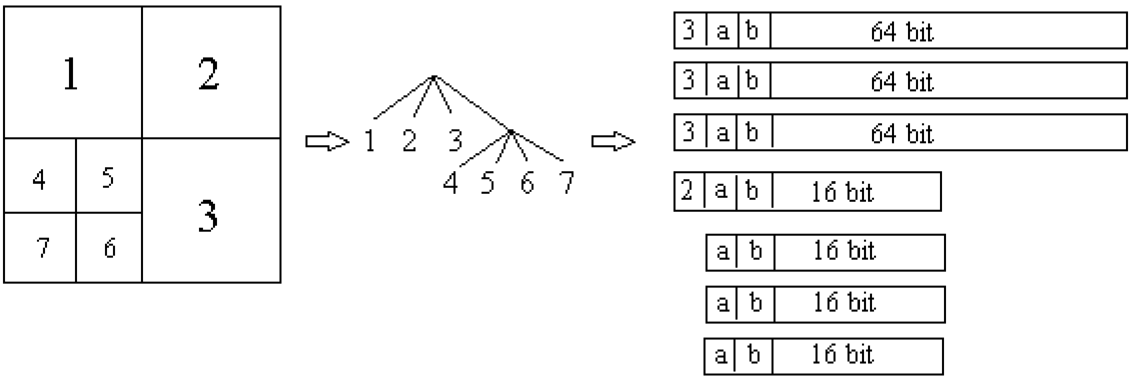
Zunächst wird jeder Block durch CCC kodiert. Hat dieser Block bereits minimale Blockgröße von 4×4 Pixeln, endet die Bearbeitung des Blockes an dieser Stelle.

Erfahrungswerte haben gezeigt, daß sich eine weitere Unterteilung nicht lohnt. Ansonsten wird Δe nach der folgenden Formel für den Block berechnet:

$$\Delta e = \sum_B \| \; p - p' \; \| .$$

Hierbei wird die Euklidnorm der Differenz des ursprünglichen Pixel-Farbvektors und des vorläufig neuen Farbvektors für den gesamten Block aufaddiert. Falls Δe kleiner ist als der Schwellenwert, wird mit der Bearbeitung des Blocks an dieser Stelle abgebrochen. Sonst wird der Block in vier Subblöcke unterteilt, und das Verfahren für jeden der Subblöcke rekursiv angewendet.

Im Gegensatz zu CCC, dessen Kompressionseinheiten ohne weitere Strukturinformation in den Datenstrom eingereiht werden konnten, handelt es sich bei der Ausgabe von XCCC um einen Baum, dessen Knoten jeweils vier Nachfolger besitzen. Diese Datenstruktur repräsentiert die Hierarchie der Blöcke bzw. Subblöcke. Deshalb ist es hier nötig, die Ausgaben mit einer komplexeren Strukturinformation zu versehen.
Das folgende Beispiel verdeutlicht die Behandlung eines 16×16 Blocks:



Für jeden Block wird dies durch Anfügen eines Etiketts oder Abzeichens vollzogen.
Dieses Etikett enthält den Wert des Logarithmus zur Basis 2 der Kantenlänge eines kodierten Blocks.
Blöcke mit minimaler Größe, also Kantenlänge vier, benötigen nur ein Etikett für vier Blöcke, weil ein minimaler Block nur drei weiteren dieser Größe vorausgehen kann. Die Ausgaben pro Block setzen sich also aus dem jeweiligen Etikett, den Indizes und der Bitmatrix zusammen. Für die Blöcke eins bis drei mit der Kantenlänge acht ergibt sich als Etikett der Wert drei, wenn man den Logarithmus zur Basis 2 auf acht anwendet. Der Subblock Nummer vier mit der Kantenlänge vier erhält demnach einen Etikettenwert von zwei. Da er die minimale Blockgröße besitzt, folgen ihm laut Definition drei weitere minimale Blöcke, bei denen auf das Etikett verzichtet werden kann. Es genügt also das Etikett mit dem Wert zwei für die vier minimalen Subblöcke.

3 Implementation

Die Darstellung der Objektbeziehungen und Nachrichtenübermittlungen zwischen den Objekten in Abbildung 3.1 kann aufgrund des komplexen Zusammenspiels innerhalb der Objekte nicht zum alleinigen Verständnis der Funktionsweise dienen. Sie bietet jedoch eine ergänzende Orientierung im Laufe der folgenden ausführlichen Erklärung.

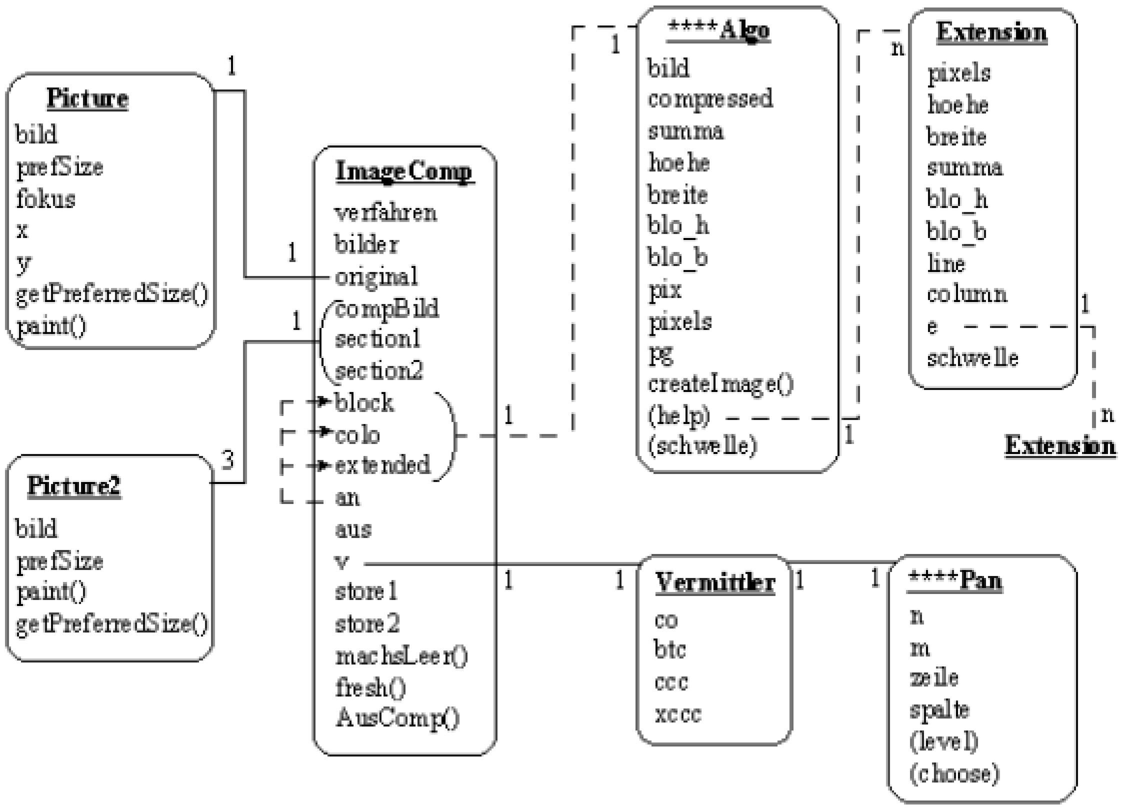


Abbildung 3.1 Objektmodell

Die gestrichelten Pfeile deuten an, daß die Zeigervariablen ‘block, colo und extended’ der Hauptappletklasse ImageComp eigentlich nur dann jeweils ein BTCAlgo-, CCCAlgo- oder XCCCAlgo-Objekt referenzieren, wenn die an-Checkbox ein Ereignisobjekt erzeugt hat. Existiert ein von ‘extended’ referenziertes XCCCAlgo-Objekt, wurde also vom User das komplizierteste der drei implementierten Verfahren ausgewählt und die Kompression gestartet, ist dieses gegenüber den beiden anderen um die Attribute ‘help’ und ‘schwelle’ reicher. Die Zeigervariable ‘help’ zeigt dann eventuell, dies ist abhängig von den zugrundeliegenden Bildgegebenheiten, auf ein Extension-Objekt, das selbst eine Referenz auf ein anderes Objekt der eigenen Klasse beinhalten kann. In jedem Fall gehört die oben beschriebene Objektkonstellation nicht zum statischen Laufzeitzustand des Applets, sondern läuft im Rahmen der Ereignisverarbeitung ab, die ihren Ursprung im Zustand der an-Checkbox findet.

3.1 Die Initialisierung - init()

Für das Applet wurde ein BorderLayout gewählt, in dem die „Untercontainer"-Panels p1 bis p5 ihren Platz finden. Auf das genaue Vorgehen hierbei sowie auf die Beschreibung diverser Label-Objekte wird an dieser Stelle aus Gründen der Übersichtlichkeit verzichtet. Da die Einträge des Choice-Objekts 'bilder' von dem gewählten Eintrag des Choice-Objekts 'verfahren' abhängen, werden 'bilder' zunächst nur die Bildeinträge des ersten Kompressionsverfahrens BTC beigefügt. Im Zusammenhang mit dem Laden und Anzeigen des ersten Eintrags aus 'bilder' ist erwähnenswert, daß dieser Eintrag mit dem Namen der zugrundeliegenden Bilddatei, ohne Formatangabe, übereinstimmt. Die Abfrage des gewählten 'bilder'-Eintrags wird zunächst in einem String zwischengespeichert. Dieser String fließt mit der Angabe des Ordners, in dem die Bilddatei enthalten ist, und der expliziten Formatangabe in die Ermittlung der Pfadangabe relativ zum URL der HTML-Seite ein, in der die Applet-Markierung enthalten ist. Die Methode 'getCodeBase()', die diesen URL liefert, und die Pfadangabe stellen nun die Argumente der 'getImage()'-Methode dar, welche selbst ein Image-Objekt als Ergebniswert zurückliefert. Dieses Image-Objekt fließt in die Konstruktion des Picture-Objekts ein, das von der Zeigervariable 'original' referenziert wird.

Das eigentliche Laden des Bildes findet also erst während dieser Konstruktion durch ein MediaTracker-Objekt statt. Das Picture-Objekt ist eine Subklasse der Klasse Component. Nach dem Ladevorgang werden die Bildmaße an das Dimension-Objekt 'prefSize' übergeben, das außerdem den Ergebniswert der Methode 'getPreferredSize()' darstellt. Die gewünschte „natürliche" Größe des Picture-Objekts wird durch Überschreiben seiner ' getPreferredSize()'-Methode auf Klassenebene festgelegt. In der Zielsetzung der Ausschnittmarkierung wird ein MouseListener von Belang sein, der bei dem Picture-Objekt registriert wird. Nach Einfügen der von 'original' referenzierten 'Picture'-Komponente in das Applet wird seine paint()-Methode aufgerufen. Sie zeichnet nicht nur das geladene Bild auf die Komponente, sondern zusätzlich ein gelbes Rechteck mit dem Ursprungspunkt (x,y). Zu diesem Zeitpunkt sind diese Variablen lediglich mit dem Wert null initialisiert. Es wird also deutlich, daß zumindest immer ein Bildausschnitt mit dem Ursprung (0,0) im Applet enthalten sein soll. Sobald die 'original'-Komponente angezeigt wird, werden ihre Bildmaße abgefragt und daraus der Speicherbedarf ermittelt. Dieser wird im TextField-Objekt 'store1' angezeigt. Der erste Schritt zur Anzeige des Ausschnitts ist der CropImageFilter. Die Argumente, die in seinen Konstruktor einfließen, beschreiben das Rechteck, das aus der 'original'-Komponente ausgeschnitten werden soll. Über ein FilteredImageSource-Objekt wird der ImageProducer des Originalbilds mit dem CropImageFilter zu einem neuen ImageProducer verknüpft. Dieser neue ImageProducer bildet das Argument der createImage()-Methode, die den Ausschnitt als Image-Objekt liefert. Auf diesen Ausschnitt wird die Image-Methode getScaledInstance() angewendet, die ihn auf 100x100 Pixel vergrößert. Das Resultat ist erneut ein Image-Objekt. Es stellt das Konstruktorargument eines Picture2-Objektes dar, das von 'section1' referenziert wird. Nun folgt die Erzeugung zweier weiterer Picture2-Objekte, die von 'compBild' und 'section2' referenziert werden. Dieses läuft analog zur Erzeugung der 'original'- und 'section1'-Komponenten ab. Beide neuen Komponenten stellen vorerst nur Platzhalter für die komprimierten Bilder da, werden also mit „leeren", schwarzflächigen Bildern gezeichnet. Es wird auf die Erzeugung einer CheckboxGroup hingewiesen, der die Checkbox-Objekte 'an' und 'aus' angehören. Die 'aus'-Checkbox ist zu Beginn aktiviert.

Die Abfrage des in ‘verfahren’ gewählten Eintrags stellt das Konstruktorenargument des Vermittler-Objekts dar, das von ‘v’ referenziert wird. Es handelt sich um eine Subklasse der Klasse Panel. Im Rahmen seiner Konstruktion macht der gewählte Eintrag den Ausschlag, ob ein BTCPan-, CCCPan- oder XCCCPan-Objekt erzeugt und dem Vermittler-Objekt als Container hinzugefügt wird. Innerhalb deren Konstruktion werden die Eingabeobjekte für den User erzeugt. Im Falle des BTCPan-Objekts, denn „Block Truncation Code“ ist der erste Eintrag in ‘verfahren’, handelt es sich um zwei Scrollbar-Objekte ‘n’ und ‘m’, die dem User die Möglichkeit bieten, die Blockgrößen beliebig zu wählen. In den beiden TextField-Objekten ‘zeile’ und ‘spalte’ werden die in den Scrollbars eingestellten Werte angezeigt.

Nach der Registration verschiedener Listener-Klassen bei den Objekten ‘verfahren’, ‘bilder’, ‘an’ und ‘aus’ sowie zweier anonymer FocusListener, die zusätzlich an den beiden Checkbox-Objekten zur Verarbeitung von low-level-Events angebracht werden, ist die Initialisierung abgeschlossen.

3.2 Die inneren Listener-Klassen der Hauptappletklasse ‘ImageComp’

3.2.1 CompressionListener

Dieser Listener wurde beim Choice-Objekt ‘verfahren’ registriert. Nachdem der Zustand der ‘aus’-Checkbox aktiviert wurde, werden die ‘ImageComp’-Instanzmethoden machsLeer() und fresh() aufgerufen. Die Methode machsLeer() überschreibt im TextField ‘store2’ den Speicherbedarf des komprimierten Bildes mit „0 Bytes“. Danach entfernt sie die ‘compBild’-Komponente aus dem Container. Sie wird neu erzeugt, diesmal aber wie in der Initialisierung mit dem schwarzflächigen, leeren Bild. Anschließend wird mit der ‘section2’-Komponente ebenso verfahren. Danach wird der nächstumgebende Container neu arrangiert. Die Methode fresh() hat die Aufgabe, das Choice-Objekt ‘bilder’ mit den zum gerade gewählten Kompressionsverfahren gehörenden Bildeinträgen zu aktualisieren. Die Abfrage nach dem ersten nun in ‘bilder’ ausgewählten Eintrag wird in einem String-Objekt zwischengespeichert. Über die Argumente getCodeBase() und die Pfadangabe der getImage()-Methode wird das zugrundeliegende Image-Objekt wie bereits vorgestellt neu ermittelt. Dieses wird dem Konstruktor eines neuen Picture-Objekts, das wieder von der Zeigervariablen ‘original’ referenziert wird, übergeben. Nach Ermittlung und Anzeige des Speicherbedarfs der neuen ‘original’-Komponente in ‘store1’ wird das neue Originalbild im Applet angezeigt. Die Filtermethodik aus der Initialisierung wird auf das neue Bild angewendet und mit dem Resultat eine neue ‘section1’-Komponente erzeugt und hinzugefügt.

Eine weitere wichtige Aufgabe des CompressionListeners ist die Erzeugung der zum neu gewählten Verfahren gehörenden Eingabemöglichkeiten für den User. Die Abfrage nach dem eingestellten Eintrag in ‘verfahren’ stellt wieder das Konstruktorargument für ein neues Vermittler-Objekt dar, das zusätzlich während seiner Erzeugung das entsprechende ****Pan-Objekt aufbaut und aufnimmt.

3.2.2 BilderListener

Der BilderListener wurde beim Choice-Objekt 'bilder' registriert. Nach der Aktivierung der 'aus'-Checkbox wird lediglich die Methode machsLeer() aufgerufen, um das komprimierte Bild und den Ausschnitt zu „löschen". Die Aktualisierung von Bildeinträgen durch fresh() macht auch keinen Sinn, da das Verfahren selbstverständlich beibehalten wurde. Da die Erzeugung eines neuen verfahrensspezifischen Eingabeobjekts unnötig ist, wird kein neues Vermittler-Objekt erzeugt. Ansonsten sind die beiden Listener in ihrer Ereignisverarbeitung identisch. Auch hier werden die neuen 'original'- und 'section1'-Komponenten mit demselben Vorgehen erzeugt und angezeigt.

3.2.3 AusListener

Der AusListener wurde bei der 'aus'-Checkbox registriert. Der User hat damit die Möglichkeit, die Kompression explizit auf „Aus" zu stellen. In der Regel wird das nicht nötig sein, da die Kompression nicht nur in den bereits vorgestellten, sondern auch in den noch folgenden Listener-Klassen automatisch auf „Aus" gestellt wird. Das komprimierte Bild und sein Ausschnitt werden entfernt und neue 'compBild'- und 'section2'-Komponenten mit schwarzflächigen Bildern erzeugt. Die Anzeige des Speicherbedarfs in 'store2' wird zurückgesetzt.

3.2.4 AnListener

Der AnListener wurde bei der 'an'-Checkbox registriert. Der Index des eingestellten Verfahrens in 'verfahren' wird überprüft. Danach wird die entsprechende if-Anweisung ausgeführt. Eine ausführliche Fallbehandlung erfolgt am Beispiel von 'Color Cell Compression'. Es wird ein CCCAlgo-Objekt erzeugt, das von der Zeigervariablen 'colo' der Hauptappletklasse referenziert wird. An seinen Konstruktor werden fünf Argumente übergeben:
Das Image-Objekt und seine Bildmaße, die in der 'original'-Komponente gekapselt sind, ferner die Werte der Scrollbars innerhalb von CCCPan, das aufgrund des eingestellten Verfahrens die einzige Komponente des Vermittler-Containers ist. Damit fließen die vom User gewählten Blockgrößen in die Konstruktion des CCCAlgo-Objekts ein. Nachdem dessen Instanzvariablen mit den Konstruktorargumenten initialisiert wurden, werden innerhalb des Konstruktors die vier Methoden first(), second(), third() und fourth() aufgerufen. Das CCCAlgo-Objekt ist erst nach Abarbeitung der fourth()-Methode vollständig erzeugt. Innerhalb der first()-Methode bewirkt die Erzeugung eines PixelGrabber-Objekts, daß die Pixelwerte des übergebenen Image-Objekts als int-Werte in ein eindimensionales 'pix'-Array transformiert werden. Dieser int-Wert gliedert sich in jeweils 8 Bit für Transparenz, Rot, Grün und Blau. Danach werden die Komponenten des pix-Array in ein zweidimensionales 'pixels'-Array übertragen, dessen iterative Behandlung bei der Pixelverarbeitung günstiger ist. In der Methode second() kommt es zur Anwendung des CCC-Algorithmus. Nach der Deklaration von verfahrensspezifischen Lokalvariablen gliedert sie sich in ein globalumgebendes for-Schleifenpaar, dessen Aufgabe es ist, die Startkoordinaten der Blöcke zu ermitteln, also des Pixels, das sich

ganz links oben in einem Block befindet. Nur diejenigen Koordinaten werden für einen „großen“ Schleifendurchlauf akzeptiert, die zu einem Block gehören, der mit seiner kompletten Größe vollständig ins Bild „paßt“. Innerhalb dieses globalen for-Schleifenpaars wird ein „kleineres“ for-Schleifenpaar häufig wiederholt, es hat die Aufgabe, alle Pixel eines Blocks spalten- und zeilenweise zu durchlaufen. Am Ende jedes Durchlaufs wird die Startkoordinate des Blockes rekonstruiert, um einen erneuten Durchlauf möglich zu machen. Zunächst werden die Helligkeitseindrücke der Pixel eines Blockes aufaddiert, um den Blockmittelwert zu ermitteln. Ein zweiter „Blockdurchlauf“ zählt die helleren und dunkleren Pixel, indem der Helligkeitseindrucks des Pixels mit dem Blockmittelwert verglichen wird. Falls das Pixel zu den dunkleren gehört, werden die Werte seiner Farbanteile in den Lokalvariablen ‘rotfora’, ‘grfora’ und blfora’ getrennt aufaddiert, andernfalls in ‘rotforb’, ‘grforb’ und ‘blforb’. Nach „Herausfiltern“ des entsprechenden Farbanteils aus dem Pixelwert wird vor der Addition im Falle von Rot ein 16-Bit-Shift, im Falle von Grün ein 8-Bit-Shift nach rechts durchgeführt. Durch Division der sechs Summen durch die Anzahl der helleren bzw. dunkleren Pixel werden die zwei endgültigen Farbwerte des Blocks berechnet und in den Lokalvariablen ‘arot’, agruen’, ‘ablau’ bzw. ‘brot’, bgruen’, ‘bblau’ gespeichert.

Im letzten Blockdurchlauf erfolgt die Zuweisung der beiden Farbwerte an die pixels-Array-Plätze, abhängig vom Vergleich des Helligkeitseindrucks des Pixels zum Blockmittelwert. Der Zuweisung geht die logische Veroderung von vier Operanden voraus. Der ursprüngliche Transparenzwert des Pixels wird beibehalten, ‘arot’ bzw. ‘brot’ werden zuvor um 16 Bits nach links geshiftet, ‘agruen’ bzw. ‘bgruen’ um 8 Bit, ‘ablau’ bzw. ‘bblau’ fließen direkt in die Veroderungsoperation ein. Vor Abschluß einer Blockverarbeitung wird der Speicherbedarf des komprimierten Blockes berechnet und in der Instanzvariablen ‘summa’ aufaddiert. Die Methode third() transportiert die neu berechneten Pixelwerte aus dem zweidimensionalen pixels-Array in den eindimensionalen pix-Array. In der Methode fourth() sorgt die Erzeugung eines MemoryImageSource-Objekts dafür, daß die Daten des pix-Array an einen ImageProducer als Bilddaten übergeben werden. Es resultiert hier ein ImageProducer, der als Argument an die createImage()-Methode übergeben wird. Deren Ergebniswert ist ein Image-Objekt, das das komprimierte Bild repräsentiert. Es wird von der CCCAlgo-Instanzvariablen ‘compressed’ referenziert. Da die Methode fourth() aus dem Konstruktor von CCCAlgo aufgerufen wird, ist dessen Erzeugung mit der Ermittlung des komprimierten Bildes abgeschlossen.

In der weiteren Ereignisbehandlung des AnListeners der Hauptappletklasse wird auf das von ‘compressed’ referenzierte Image-Objekt zugegriffen und damit eine neue ‘compBild’-Komponente erzeugt und ins Applet aufgenommen. Das TextField-Objekt ‘store2’ zeigt den in CCCAlgo gekapselten Speicherbedarf des komprimierten Bildes an. Um die Folgen der Kompression auf den in der ‘section1’-Komponente dargestellten Bildausschnitt zu veranschaulichen, erfolgt der Aufruf der Methode AusComp() der Hauptappletklasse. Mittels der vorgestellten Filtermethodik wird der in ‘original’ markierte Ausschnitt mit denselben Ursprungskoordinaten aus der ‘compBild’-Komponente ausgeschnitten und auf ‘section2’ vergrößert dargestellt. Die Ereignisbehandlung für ‘Extended Color Cell Compression’ unterscheidet sich in programmtechnischen und verfahrensspezifischen Aspekten. Zur Erzeugung eines XCCCAlgo-Objektes ist ein weiteres Konstruktorargument notwendig: Der Schwellenwert. Sein Wert wird aus dem TextField-Objekt ‘level’ entnommen, das zu den Komponenten von XCCCPan gehört.

Während sich BTCAlgo und CCCAlgo ähneln, weicht die Struktur des XCCCAlgo-Objekts verfahrensbedingt ab. In XCCCAlgo ist die Zeigervariable ‘help’ als Referenz auf ein Extension-Objekt deklariert. Bis auf die Konstruktorargumenterweiterung durch den Schwellenwert, der an die Variable ‘schwelle’ übergeben wird, unterscheidet sich das XCCCAlgo-Objekts von dem CCCAlgo-Objekts in der Methode second().Im ersten Blockdurchlauf werden die Helligkeitseindrücke der Pixel aufaddiert und daraus der Blockmittelwert berechnet. Es folgt die Ermittlung der beiden Farbwerte, die für den gerade bearbeiteten Block in Betracht kommen. Der folgende Blockdurchlauf dient dagegen der Ermittlung von Δe! Die Euklidnorm der Differenz zwischen dem ursprünglichen Pixelwert und seinem provisorisch neuen wird in der Lokalvariablen ‘delta’ aufaddiert. Welcher der beiden ermittelten Farbwerte der provisorisch neue des Pixels ist, hängt selbstverständlich wieder vom Vergleich des Helligkeitseindrucks des Pixels mit dem Mittelwert ab. Falls eines der beiden Blockmaße kleiner als acht Pixel ist, sich also eine Unterteilung des Blocks in vier Subblöcke erfahrungsgemäß nicht lohnen würde, oder Δe in seinem Wert kleiner als der übergebene Schwellenwert ist, so erfolgt in einem abschließenden Blockdurchlauf die Zuweisung der beiden ermittelten Farbwerte an die Pixel des Blockes. Wenn beide Blockmaße einen Wert von mindestens acht Pixeln betragen und Δe in seinem Wert größer oder gleich dem Schwellenwert ist, wird ein Extension-Objekt erzeugt, das von ‘help’ referenziert wird. Ein Extension-Objekt ist eine „schlanke“ Version von XCCCAlgo! Seinem Konstruktor werden die Startkoordinaten des Blockes, dessen vollständige Blockverarbeitung gerade in XCCCAlgo scheiterte, dessen Blockmaße als neue „Bildhöhe“ und „Bildbreite“, die halbierten Blockmaße als neue Blockmaße, eine Referenz auf das pixels-Array und der Schwellenwert übergeben. Das Extension-Objekt behandelt diesen Block als vollständiges Bild, es „weiß“ nicht, daß es außer diesem „Block“ noch andere Bildbereiche gibt. ‘Extension’ enthält eine Referenz ‘e’, die dafür vorgesehen ist, ein anderes Objekt derselben Klasse zu referenzieren, denn in seiner einzigen Methode second() verarbeitet Extension sein „Bild“ in derselben Blockverarbeitungsweise wie XCCCAlgo! Der ehemalige XCCCAlgo-Block wird von Extension in vier Subblöcke unterteilt, von denen jeder auf das Schwellenwert-Kriterium hin überprüft wird. Ist dieses für einen Subblock erfüllt, wird ein weiteres Extension-Objekt erzeugt und durch ‘e’ referenziert, das diesen Subblock erneut in vier Subblöcke unterteilt.

Es handelt sich also um ein klassisches Backtracking über mehrere Extension-Objekte und einem XCCCAlgo-Objekt, das die Wurzel des „Baumes“ darstellt, aber jedes dieser Objekte referenziert das gleiche pixels-Array und legt die neu berechneten Pixelwerte des „eigenen“ Bildes im dafür vorgesehenen Bereich des pixels-Array ab. In einem Knoten wird darüber entschieden, ob die entsprechenden Plätze im pixels-Array eine Zuweisung mit den zwei Farbwerten erfahren oder ob der Block an einen Sohn übergeben wird, um ihn in vier Subblöcke zu unterteilen.

3.3 Die Listener der Unterobjekte im Zusammenspiel mit den anonymen Focus-Listenern der Hauptappletklasse

3.3.1 ScLauscher in BTCPan und der bei der ‘aus’-Checkbox registrierte Focus-Listener

Der ScLauscher-Listener wurde bei den Scrollbar-Objekten ‘n’ und ‘m’ registriert. Bei jeder Modifikation der beiden Scrollbars wird deren Wert abgefragt und in den zu BTCPan gehörenden TextField-Objekten ‘zeile’ und ‘spalte’ aktualisiert. Der Zustand der ‘aus’-Checkbox der Hauptappletklasse wird aktiviert und ihr wird der Fokus zugewiesen. Der FocusListener reagiert mit dem Aufruf der Methode machsLeer(). Sie bewirkt die Anzeige des Textes „0 Bytes“ im TextField ‘store2’ sowie die Neuerzeugung der ‘compBild’- und ‘section2’-Komponenten mit den leeren, schwarzflächigen Bildern.

3.3.2 ScLauscher in CCCPan und der bei der ‘aus’-Checkbox registrierte Focus-Listener

Dieser Listener gleicht in seiner Funktionsweise dem im vorigen Abschnitt beschriebenen.

3.3.3 ScLauscher und ChLauscher in XCCCPan und der bei der ‘aus’-Checkbox registrierte FocusListener

ScLauscher ist bei den Scrollbars ‘n’ und ‘m’ registriert, der ChLauscher beim Choice-Objekt ‘choose’ von XCCCPan. Sowohl bei Modifikation der Scrollbars als auch bei Wahl eines neuen Eintrags in ‘choose’ wird der Zustand der ‘aus’-Checkbox aktiviert und ihr wird der Fokus zugewiesen. Beide Listener berechnen das Produkt aus den Werten der Scrollbars und dem ‘choose’-Objekt. Der ermittelte Schwellenwert wird im TextField ‘level’ von XCCCPan angezeigt. Die Werte der Scrollbars werden durch ScLauscher in den Textfeldern ‘zeile’ und ‘spalte’ angezeigt.

3.3.4 MausBearbeiter in Picture und der bei der ‘an’-Checkbox registrierte FocusListener

Dieses Zusammenspiel dient der Ausschnittmarkierung auf der ‘original’-Komponente, des einzigen Picture-Objekts im Applet. Der MausBearbeiter wurde beim gesamten Picture-Objekt registriert. Erfolgt auf der ‘original’-Komponente eine mousePressed-Aktion, wird ihre Instanzvariable ‘fokus’ auf true gesetzt und die Mausklick-Koordinaten an ‘x’ und ‘y’ zugewiesen. Danach erhält die ‘an’-Checkbox den Fokus. Ihr FocusListener überprüft, ob der Zustand der ‘an’-Checkbox deaktiviert ist UND die in ‘original’ gekapselte ‘fokus’-Variable den Wert true enthält, denn ‘fokus’ wird innerhalb des MausBearbeiters auf ‘false’ gesetzt, sobald die Maus die ‘original’-Komponente verläßt.

Es wird ein CropImageFilter-Objekt erzeugt, dessen „ausschneidendes“ Rechteck die Ursprungskoordinaten der in ‘original’ gekapselten Variablen ‘x’ und ‘y’ enthält. Nachdem der Filter mit dem ImageProducer des in ‘original’ gekapselten Image-Objekts zu einem neuen ImageProducer verbunden wurde, wird auf dieses Resultat die createImage()-Methode angewendet, welche als Ergebniswert den Ausschnitt als Image-Objekt liefert. Nach Anwendung der Image-Methode getScaledInstance() wird mit diesem „vergrößerten“ Ausschnitt die ‘section1’-Komponente neu erzeugt, um den neu gewählten Ausschnitt anzuzeigen. Damit der User weitere Ausschnitte wählen kann, muß daraufhin der ‘aus’-Checkbox der Fokus zugewiesen werden, damit die ‘an’-Checkbox bei Fokuserhalt durch den MausBearbeiter im Picture-Objekt ein weiteres FocusEvent erzeugt. Um die neue Markierung sichtbar zu machen, schließt die focusGained()-Methode mit der Aufforderung an die ‘original’-Komponente, sich neu zu zeichnen. Da die bei diesem Vorgang zum Tragen kommende Graphics-Methode drawRect() beim Ursprungspunkt (x,y) ansetzt, steht der neuen Ausschnittmarkierung nichts mehr im Weg.

4 Präsentation des Applets

Abbildung 4.1 zeigt den Zustand des Applets direkt nach der Initialisierung:

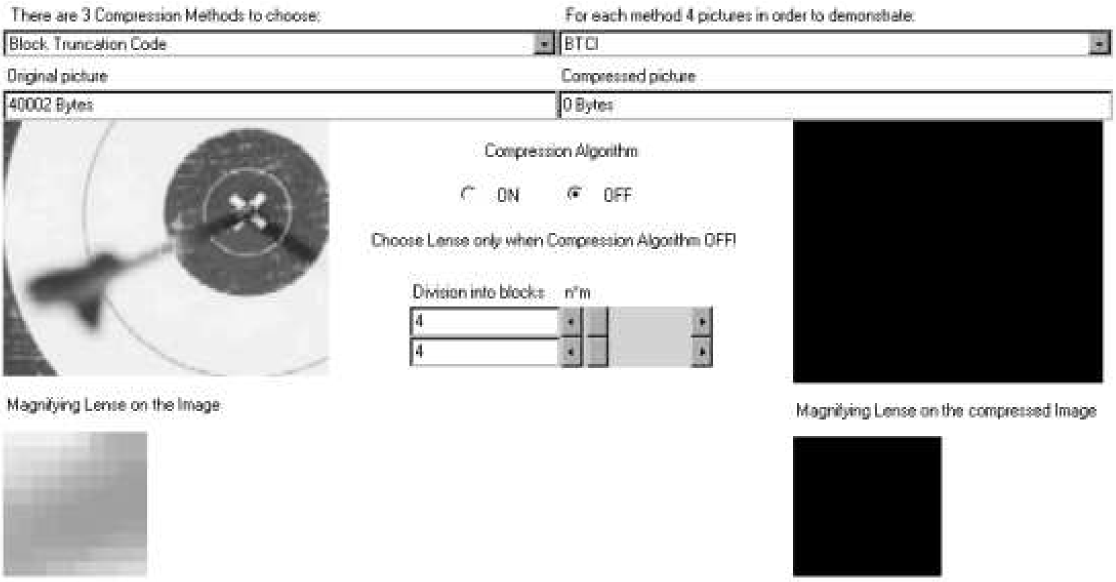


Abbildung 4.1 Applet nach Initialisierung

Der erste Eintrag im Choice-Objekt ‘verfahren’ ist ‘Block Truncation Code’. Das Choice-Objekt ‘bilder’ enthält die zu diesem Verfahren gehörenden Bildeinträge. Das zum ersten Eintrag gehörende Bild wird bereits im Applet angezeigt, außerdem ist sein Speicherbedarf aus dem darüber positionierten Textfeld ‘store1’ zu entnehmen. Ein Standardausschnitt ist auf der gleich darunter liegenden ‘section1’-Komponente dargestellt. Durch die Eingabemöglichkeiten im mittleren Bereich kann der User Einfluß auf das Kompressionsergebnis nehmen in Form von variierenden Blockangaben. Da die Kompression ausgeschaltet ist, sind rechts die schwarzflächigen Platzhalter der künftigen Kompressionen enthalten.

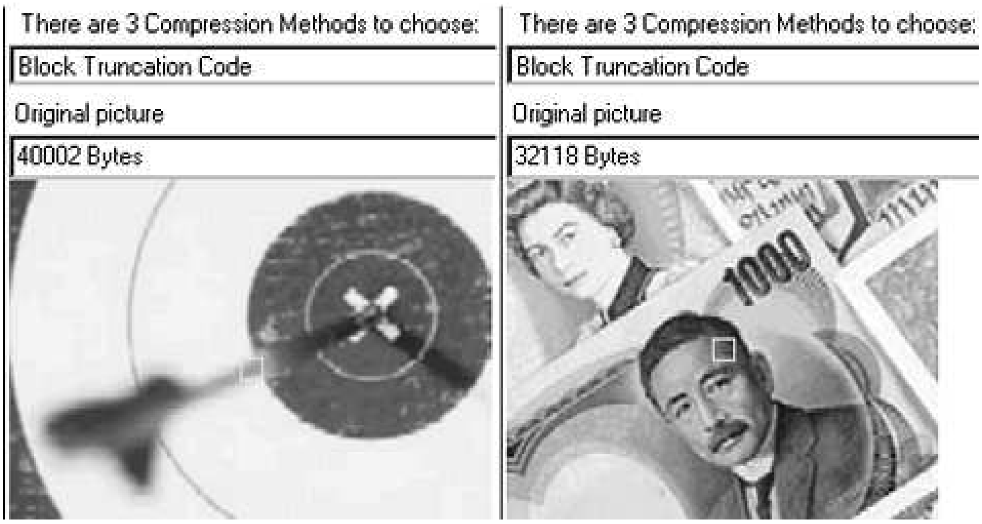


Abbildung 4.2 Anzeige des Speicherbedarfs

Sobald der User ein anderes Bild oder Verfahren wählt, was zur Neuerzeugung der ‘original’-Komponente führt, wird die Anzeige des zugehörigen Speicherbedarfs im Textfeld aktualisiert. Dieser Effekt wird in Abbildung 4.2 demonstriert.

Abbildung 4.3 zeigt die Wirkung auf das Erscheinungsbild des Applets bei Wahl eines anderen Verfahrens:

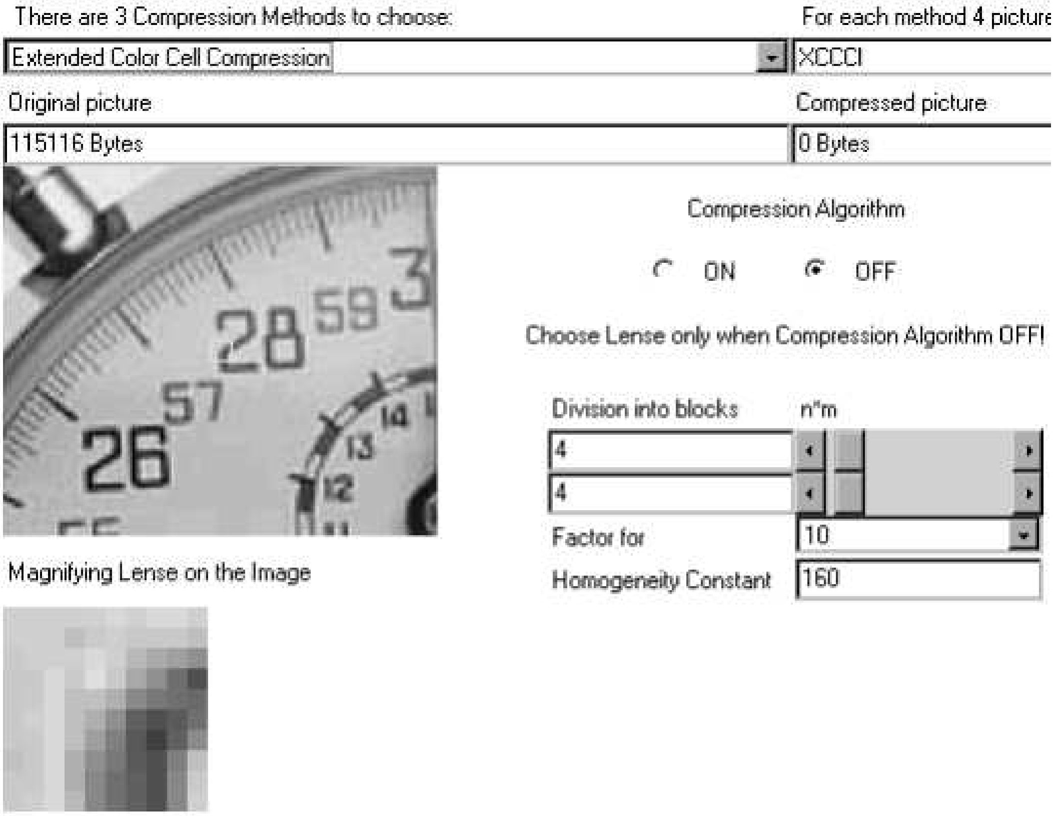


Abbildung 4.3 Modifikation des Verfahrens

Die zum neuen Verfahren gehörenden Bildeinträge werden im Choice-Objekt ‘bilder’ aufgelistet. Das erste Bild, sein Speicherbedarf und Standardausschnitt werden im Applet angezeigt. Da ‘Extended Color Cell Compression’ gewählt wurde, ist nun das komplexere Eingabeobjekt ‘XCCCPan’ im Applet enthalten.

Die Abbildung 4.4 zeigt Beispiele für die Ausschnittmarkierung. Solange die Kompression ausgeschaltet ist, hat der User die Möglichkeit, beliebige Ausschnitte des Bildes zu wählen und vergrößert auf der ‘section1’-Komponente anzeigen zu lassen. Jeder Mausklick auf der ‘original’-Komponente bewirkt, daß mit den Mausklick-Koordinaten als Ursprungspunkt ein 10x10 Pixel großes gelbes Rechteck auf das Bild gezeichnet wird. Danach wird dieser Ausschnitt um das hundertfache vergrößert auf ‘section1’ angezeigt.

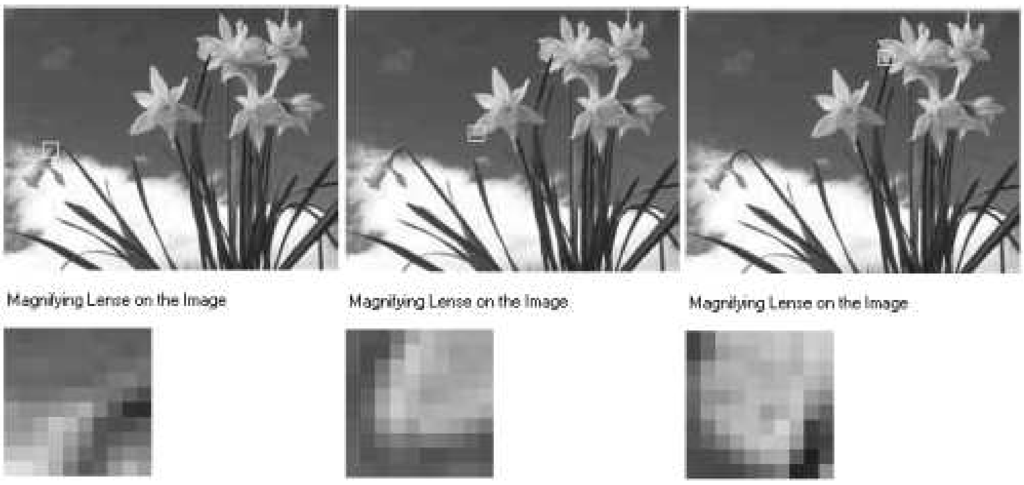


Abbildung 4.4 Ausschnittwahl und Ausschnittmarkierung

Die Abbildung 4.5 liefert vier Beispiele für das Ergebnis, das die Modifikation der Scrollbars in den Eingabeobjekten hat. Die Werte der Scrollbars werden abgefragt und in Textfeldern angezeigt. Während die slider der Scrollbars nur eine ungefähre Orientierung bezüglich der gewählten Blockdimensionen ermöglichen, erspart die direkte Anzeige ihrer Werte jede Heuristik.

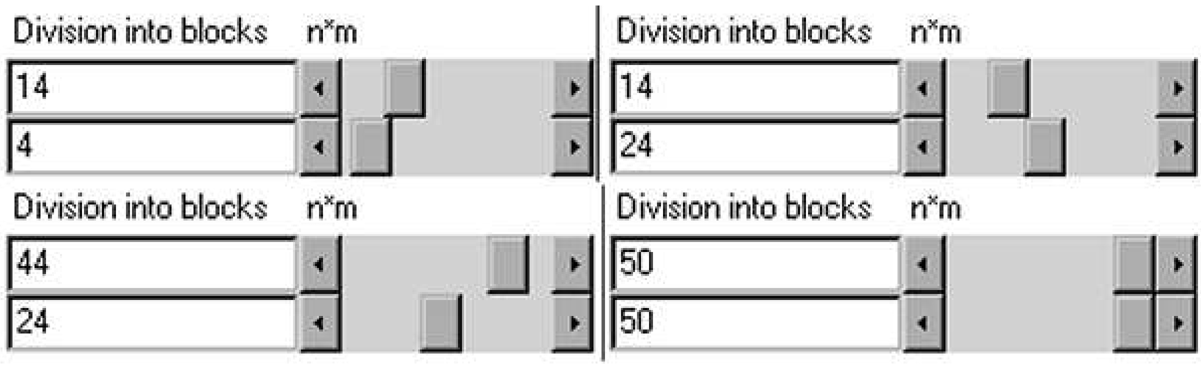


Abbildung 4.5 Modifikation der Scrollbars

Die Kompression durch „Color Cell Compression“ mit unterschiedlichen Blockgrößen wird durch die Abbildung 4.6 veranschaulicht. Für das links in der Abbildung angezeigte Kompressionsergebnis wurden die Blockgrößen 20x20 Pixel gewählt, für das rechte 50x50 Pixel. Die maximalen Blockgrößen haben einen rapiden Qualitätsverlust zur Folge, da CCC jedem Block nur zwei Farbwerte zuweist. Im Gegensatz dazu steht die bessere Kompressionsrate, die der Speicherbedarfanzeige entnommen werden kann.

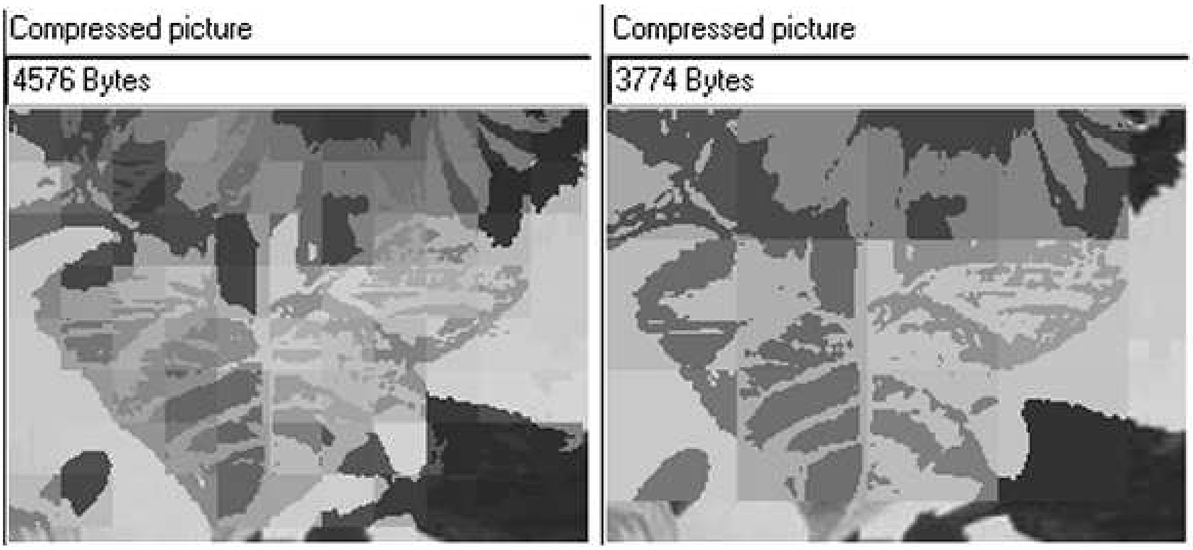


Abbildung 4.6 Kompression durch ‘Color Cell Compression’

Das mit „Homogeneity Constant“ bezeichnete Textfeld ‘level’ des Eingabeobjekts XCCCPan zeigt das Produkt aus den Blockgrößen und dem im Choice-Objekt ‘choose’ eingestellten Wert an. Sowohl bei Modifikation der Scrollbars als auch des Choice-Objekts wird der Schwellenwert neu berechnet. Der Ablauf ist in Abbildung 4.7 veranschaulicht.

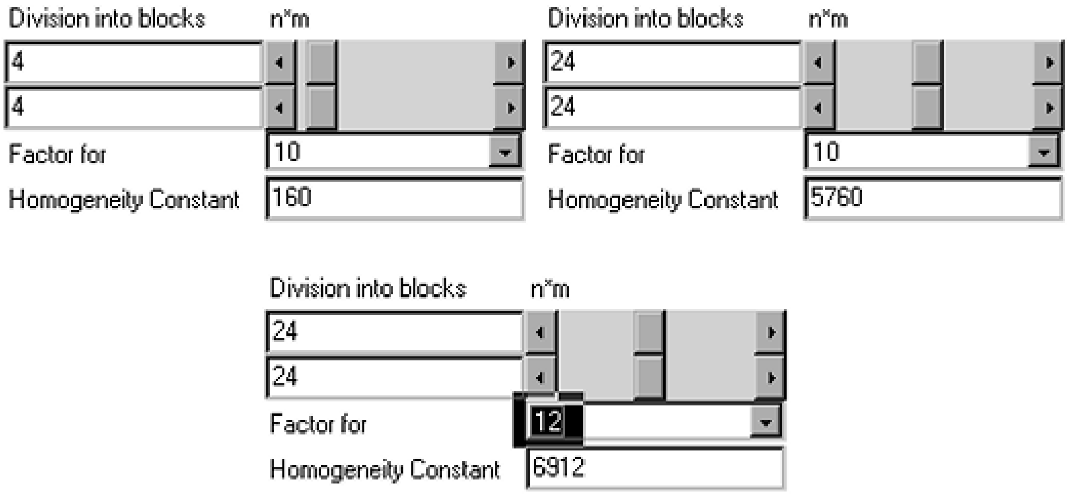


Abbildung 4.7 Berechnung des Schwellenwerts

Die nächsten beiden Abbildungen bieten einen direkten Vergleich zwischen der Kompression durch ‘Color Cell Compression’ und ‘Extended Color Cell Compression’. Abbildung 4.8 zeigt zwei Beispiele der CCC. In Abbildung 4.9 wurde dasselbe Bild mit analogen Blockgrößen gewählt und durch XCCC komprimiert. Hier unterteilen sich die Blöcke in Subblöcke. Es bilden sich feinere „Schachbrettmuster“. Die Kompressionsergebnisse sind qualitativ ansprechender. XCCC reagiert intelligenter auf starke Farbunterschiede in Teilbildbereichen, während CCC selbst bei sehr großen Blockgrößen für jeden Block nur zwei Farbwerte vorsieht.

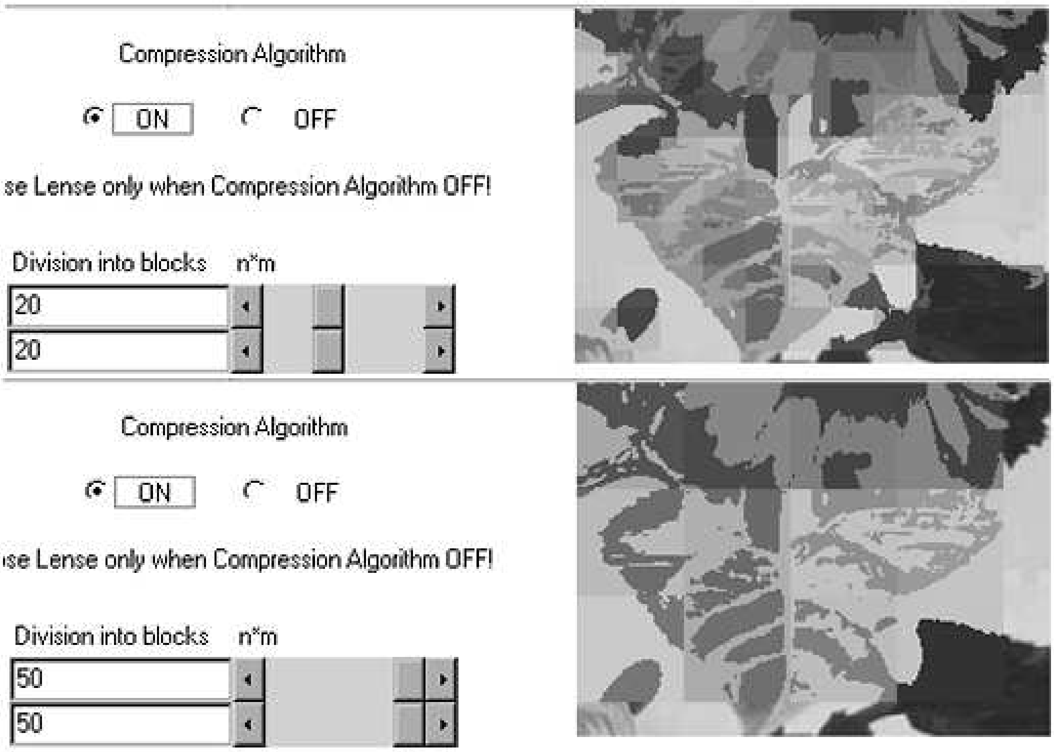


Abbildung 4.8 Zwei Beispiele der CCC

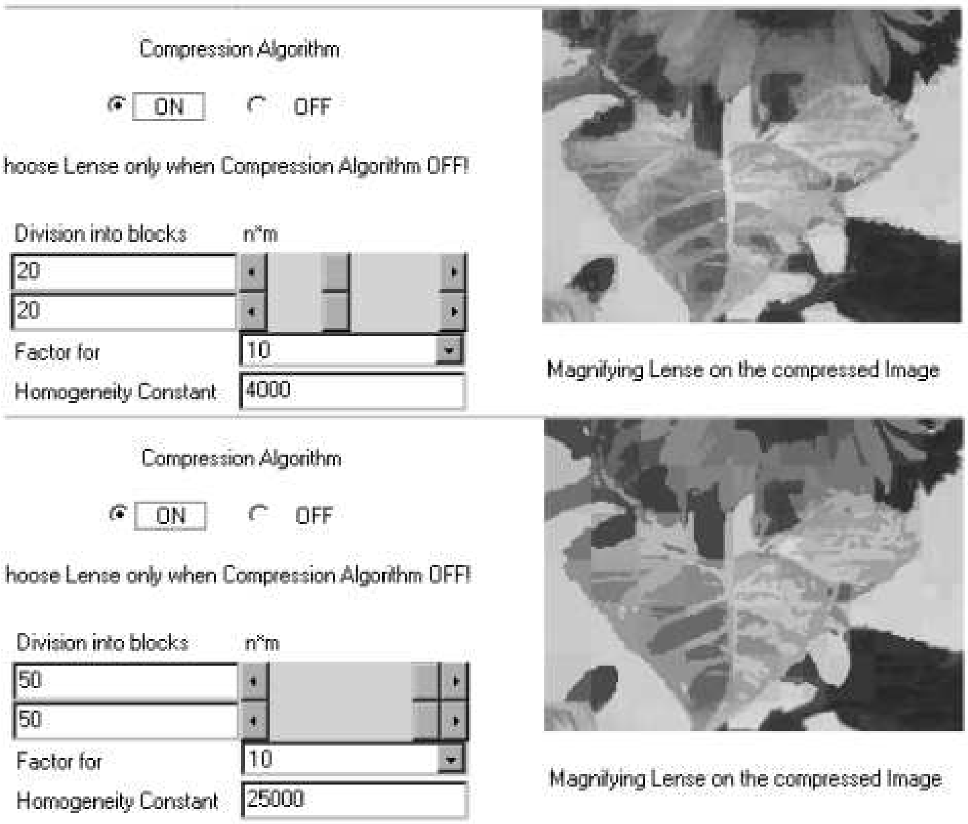


Abbildung 4.9 Zwei Beispiele der XCCC

5 Offene Punkte

Eine erneute Ausschnittwahl ist nicht mehr möglich, nachdem die ‘an’-Checkbox aktiviert, die Kompression gestartet wurde. Die Ursache liegt in der Programmtechnik, daß der MausBearbeiter der ‘original’-Komponente mit der Fokuszuweisung an die ‘an’-Checkbox endet. Die focusGained()-Methode des bei der ‘an’-Checkbox registrierten FocusListeners erzeugt dann den Ausschnitt. Ist die Kompression also durch manuelles Aktivieren der ‘an’-Checkbox bereits gestartet, hat diese damit den Fokus. Die erneute Fokuszuweisung macht sich also nicht mehr bemerkbar, es wird kein neuer Ausschnitt erzeugt. Inwieweit könnte dieser Mangel ausgeglichen werden, wenn kein manuelles Einschalten der Kompression nötig wäre? Welche Vorkehrungen sind zu treffen, um diese Prämisse zu realisieren?

CompressionListener:
Seine Aufgaben sind bisher, die zum Verfahren gehörenden Bildeinträge in ‘bilder’ zu aktualisieren und das erste dieser Bilder auf der ‘original’-Komponente darzustellen. Außerdem sorgt er für die Darstellung des Standardausschnitts durch die ‘section1’-Komponente. Zusätzlich fügt er das zum gewählten Verfahren gehörende Eingabeobjekt in das Applet ein. Er müßte nun durch die Anweisungen des AnListeners erweitert werden, diesmal allerdings mit den Standardangaben des Eingabeobjekts.

BilderListener:
Seine Aufgabe ist es, das neu gewählte Bild auf der ‘original’-Komponente in Erscheinung treten zu lassen. Auch er wird durch die Anweisungen des AnListeners ergänzt. Diesmal modifiziert durch die aktuellen User-Eingaben in dem Eingabeobjekt des zugrundeliegenden Verfahrens.

Ohne AnListener und AusListener muß die Kompression auch nach Modifikation der User-Eingaben möglich werden. Dazu müßten die Konstruktorargumente der ****Pan-Objekte eine Erweiterung durch die Referenz auf das Objekt der Hauptappletklasse erfahren. Damit versteht sich die Erweiterung der an den Eingabemöglichkeiten registrierten Listenerklassen innerhalb der ****Pan-Objekte durch die Anweisungen des AnListeners von selbst. Da die Kompression immer „eingeschaltet“ ist, ist auch die Implementation des bei der ‘aus’-Checkbox registrierten FocusListeners hinfällig geworden , dessen Aufgabe es war, die leeren Bilder als Platzhalter der künftigen Kompressionsergebnisse zu erzeugen. Welche Konsequenz haben diese neuen Bedingungen nun auf den bei der ‘an’-Checkbox registrierten FocusListener, der zum Anzeigen der neuen Ausschnitte führt?

Bei einem Mausklick auf das von ‘original’ referenzierte Picture-Objekt wird innerhalb dessen MouseListeners zunächst die ‘fokus’-Variable auf true gesetzt (bei Verlassen der Komponente durch den Mauszeiger sofort wieder auf false), die Mausklick-Koordinaten an ‘x’ und ‘y’ zugewiesen und dann der ‘an’-Checkbox der Fokus gegeben. In der focusGained()-Methode des ‘an’-FocusListeners reduziert sich die Bedingung der if-Anweisung nun auf die Prüfung, ob ‘fokus’ den Wert true enthält, die ‘an’-Checkbox den Fokus also durch Mausklick auf die ‘original’-Komponente erhalten hat. Die folgende

Filterkombination, mit der die ‘section1’-Komponente neu erzeugt wird, bleibt erhalten, ebenso die Zuweisung des Fokus an die ‘aus’-Checkbox. Letzteres ist notwendig, denn bei mehrfacher Ausschnittwahl muß die ‘an’-Checkbox den Fokus neu erhalten können, sie muß ihn zwischenzeitlich verlieren. Nach Aufforderung an die ‘original’-Komponente, sich mit aktueller Markierung neu zu zeichnen, wird die Methode AusComp() der Hauptappletklasse aufgerufen, um den gleichen Ausschnitt aus dem komprimierten Bild im Applet darzustellen.

Eine weitere Überlegung geht dahin, ohne ‘an’-FocusListener auszukommen, was bedeutet, auf die Checkbox-Objekte zu verzichten. Die Konstruktorargumente des Picture-Objekts werden um die Referenz auf das Objekt der Hauptappletklasse erweitert. Damit könnte die Neuerzeugung des Ausschnitts und die Anzeige des „komprimierten“ Ausschnitts direkt vom MausBearbeiter des Picture-Objekts übernommen werden.

Anhang: Coding in Auszügen

Zum Verständnis der Programmlogik folgt das Coding der Hauptappletklasse ‘ImageComp’, ‘Picture’ und ‘CCCAlgo’:

Coding der Klasse ‘ImageComp’

```
//  ImageComp.java

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

public class ImageComp extends Applet{

        Choice verfahren, bilder;
        Image bild = null;
        Picture original;
        Picture2 compBild, section1, section2;
        Panel p1,p2,p3,p4,p5;
        TextField store1, store2;
        BTCAlgo block;
        CCCAlgo colo;
        XCCCAlgo extended;

 public static CheckboxGroup comp;
 public static Checkbox an, aus;

        Vermittler v;


 public void init(){

        setBackground(Color.white);
        setLayout(new BorderLayout());

        p1 = new Panel();
        p1.setLayout(new GridLayout(4,2));

        verfahren = new Choice();
        bilder = new Choice();
        verfahren.add("Block Truncation Code");
        verfahren.add("Color Cell Compression");
        verfahren.add("Extended Color Cell Compression");

        bilder.add("BTCI");
        bilder.add("BTCII");
        bilder.add("BTCIII");
        bilder.add("BTCIV");

        p1.add(new Label(" There are 3 Compression Methods to choose: "));
```

```
p1.add(new Label(" For each method 4 pictures in order to demonstrate: "));
    p1.add(verfahren);
    p1.add(bilder);
    p1.add(new Label("Original picture"));
    p1.add(new Label("Compressed picture"));
    store1 = new TextField(15);
    store1.setText("0 Bytes");
    store2 = new TextField(15);
    store2.setText("0 Bytes");
    store1.setEditable(false);
    store2.setEditable(false);
    p1.add(store1);
    p1.add(store2);
    add(p1, BorderLayout.NORTH);

    p2 = new Panel();
    p2.setLayout(new GridLayout(2,1));

    String a = bilder.getSelectedItem();
    bild = getImage(getCodeBase(), "images/" + a + ".gif");
    original = new Picture(bild);
    p2.add(original);
    store1.setText(String.valueOf(original.prefSize.height*
                        original.prefSize.width)+" Bytes");
    ImageFilter filter = new CropImageFilter(0,0,10,10);
  ImageProducer prod = new FilteredImageSource(original.bild.getSource(),
                              filter);
    Image ausschnitt = createImage(prod);
  bild = ausschnitt.getScaledInstance(100,100,Image.SCALE_AREA_AVERAGING);
    section1 = new Picture2(bild);
    p2.add(section1);
    add(p2, BorderLayout.WEST);

    p3 = new Panel();
    p3.setLayout(new GridLayout(2,1));
    bild = null;
    bild = getImage(getCodeBase(), "images/leer.gif");
    compBild = new Picture2(bild);
    p3.add(compBild);
    bild = null;
    bild = getImage(getCodeBase(), "images/leer2.gif");
    section2 = new Picture2(bild);
    p3.add(section2);
    add(p3, BorderLayout.EAST);

    p4 = new Panel();
    p4.setLayout(new FlowLayout(FlowLayout.CENTER,10,10));
    p5 = new Panel();
    p5.setLayout(new FlowLayout(FlowLayout.CENTER,10,10));
    comp = new CheckboxGroup();
    an = new Checkbox("   ON   ",false, comp);
    aus = new Checkbox("   OFF   ",true, comp);
    p5.add(an);
    p5.add(aus);
    p4.add(p5);
    v = new Vermittler(verfahren.getSelectedItem());
```

```
        p4.add(v);
        add(p4, BorderLayout.CENTER);


        CompressionListener lisc = new CompressionListener();
        verfahren.addItemListener(lisc);

        BilderListener lisb = new BilderListener();
        bilder.addItemListener(lisb);

        AusListener lisaus = new AusListener();
        AnListener lisan = new AnListener();
        aus.addItemListener(lisaus);
        an.addItemListener(lisan);

        aus.addFocusListener(new FocusAdapter(){
                    public void focusGained(FocusEvent e){
                        machsLeer();
                    }
                    });

        an.addFocusListener(new FocusAdapter(){
         public void focusGained(FocusEvent e){

         if( (an.getState()==false) && (original.fokus) ){
           ImageFilter filters =
               new CropImageFilter(original.x, original.y, 10, 10);
           ImageProducer producer =
               new FilteredImageSource(original.bild.getSource(), filters);
           Image ausschneid = createImage(producer);
           Image pics =
            ausschneid.getScaledInstance(100,100,Image.SCALE_AREA_AVERAGING);
           p2.remove(section1);
           section1 = null;
           section1 = new Picture2(pics);
           p2.add(section1);
           p2.validate();
           aus.requestFocus();
           original.repaint();

         }

        }
    });

}  //init


public void update(Graphics g){
        super.paint(g);
}

void AusComp(){
     ImageFilter fil =
          new CropImageFilter(original.x, original.y, 10, 10);
     ImageProducer pr =
```

```
            new FilteredImageSource(compBild.bild.getSource(), fil);
      Image au = createImage(pr);
      Image pi =
            au.getScaledInstance(100,100,Image.SCALE_AREA_AVERAGING);
      p3.remove(section2);
      section2 = null;
      section2 = new Picture2(pi);
      p3.add(section2);
      p3.validate();
}

void machsLeer(){
            store2.setText("0 Bytes");
            p3.remove(compBild);
            compBild = null;
      compBild = new Picture2(getImage(getCodeBase(),"images/leer.gif"));
            p3.add(compBild);
            p3.remove(section2);
            section2 = null;
      section2 = new Picture2(getImage(getCodeBase(), "images/leer2.gif"));
            p3.add(section2);
            p3.validate();
}

void fresh(){

       bilder.removeAll();

       if(verfahren.getSelectedIndex()==0){

          bilder.add("BTCI");
          bilder.add("BTCII");
          bilder.add("BTCIII");
          bilder.add("BTCIV");
       };

       if(verfahren.getSelectedIndex()==1){

          bilder.add("CCCI");
          bilder.add("CCCII");
          bilder.add("CCCIII");
          bilder.add("CCCIV");
       };

       if(verfahren.getSelectedIndex()==2){

          bilder.add("XCCCI");
          bilder.add("XCCCII");
          bilder.add("XCCCIII");
          bilder.add("XCCCIV");
       };
}


class CompressionListener implements ItemListener{
      public void itemStateChanged(ItemEvent e){
```

```
            aus.setState(true);
            machsLeer();

            fresh();

            String x = bilder.getSelectedItem();
            p2.remove(original);
            p2.remove(section1);
            original = null;
            section1 = null;
            if(verfahren.getSelectedIndex() == 0){
    original = new Picture(getImage(getCodeBase(),"images/" + x + ".gif"));
    store1.setText(String.valueOf(original.prefSize.height*
                         original.prefSize.width)+" Bytes");
            }
            else{
    original = new Picture(getImage(getCodeBase(),"images/" + x + ".jpg"));
    store1.setText(String.valueOf(original.prefSize.height*
                         original.prefSize.width*3)+" Bytes");
            }

            p2.add(original);

            ImageFilter filter = new CropImageFilter(0,0,10,10);
            ImageProducer prod =
              new FilteredImageSource(original.bild.getSource(), filter);
            Image ausschnitt = createImage(prod);
         Image pic =
           ausschnitt.getScaledInstance(100,100,Image.SCALE_AREA_AVERAGING);
            section1 = new Picture2(pic);
            p2.add(section1);
            p2.validate();

            p4.remove(v);
            v = null;
            v = new Vermittler(verfahren.getSelectedItem());
            p4.add(v);
            p4.validate();
      }
}


class BilderListener implements ItemListener{
      public void itemStateChanged(ItemEvent e){

           aus.setState(true);
           machsLeer();

           String y = bilder.getSelectedItem();
           p2.remove(original);
           p2.remove(section1);
           original = null;
           section1 = null;
           if(verfahren.getSelectedIndex() == 0){
    original = new Picture(getImage(getCodeBase(),"images/" + y + ".gif"));
```

```
    store1.setText(String.valueOf(original.prefSize.width*
                          original.prefSize.height)+" Bytes");
          }
          else{
    original = new Picture(getImage(getCodeBase(),"images/" + y + ".jpg"));
    store1.setText(String.valueOf(original.prefSize.height*
                          original.prefSize.width*3)+" Bytes");
          }

          p2.add(original);
          ImageFilter filter =
             new CropImageFilter(0,0,10,10);
          ImageProducer prod =
             new FilteredImageSource(original.bild.getSource(), filter);
          Image ausschnitt = createImage(prod);
         Image pic =
          ausschnitt.getScaledInstance(100,100,Image.SCALE_AREA_AVERAGING);
          section1 = new Picture2(pic);
          p2.add(section1);
          p2.validate();
      }
}


class AusListener implements ItemListener{
      public void itemStateChanged(ItemEvent e){
          if(aus.getState()==true){
             p3.remove(compBild);
             compBild = null;
        compBild = new Picture2(getImage(getCodeBase(),"images/leer.gif"));
             p3.add(compBild);
             store2.setText("0 Bytes");
             p3.remove(section2);
             section2 = null;
        section2 = new Picture2(getImage(getCodeBase(),"images/leer2.gif"));
             p3.add(section2);
             p3.validate();
          };

      }
}

class AnListener implements ItemListener{
      public void itemStateChanged(ItemEvent e){

          if(an.getState() == true){

             if(verfahren.getSelectedIndex() == 0){

                block = new BTCAlgo(original.bild,
                            original.prefSize.height,
                            original.prefSize.width,
                            v.btc.n.getValue(),
                            v.btc.m.getValue());
                p3.remove(compBild);
                compBild = null;
```

```
            compBild = new Picture2(block.compressed);
            p3.add(compBild);
            p3.validate();
            block = null;
            AusComp();

          }
          if(verfahren.getSelectedIndex() == 1){

            colo = new CCCAlgo(original.bild,
                          original.prefSize.height,
                          original.prefSize.width,
                          v.ccc.n.getValue(),
                          v.ccc.m.getValue());
            p3.remove(compBild);
            compBild = null;
            compBild = new Picture2(colo.compressed);
            p3.add(compBild);
            p3.validate();
            store2.setText(String.valueOf(colo.summa/8)+" Bytes");
            colo = null;
            AusComp();
          }
          if(verfahren.getSelectedIndex() == 2){

        extended = new XCCCAlgo(original.bild,
                          original.prefSize.height,
                          original.prefSize.width,
                          v.xccc.n.getValue(),
                          v.xccc.m.getValue(),
                         Integer.parseInt(v.xccc.level.getText()));
            p3.remove(compBild);
            compBild = null;
            compBild = new Picture2(extended.compressed);
            p3.add(compBild);
            p3.validate();
            store2.setText(String.valueOf(extended.summa/8)+" Bytes");
            extended = null;
            AusComp();
          }

        }
      }
 }

} //Hauptapplet-klasse
```

Coding der Klasse 'Picture'

```
// Picture.java

import java.awt.*;
import java.awt.event.*;

public class Picture extends Component{

      Image bild = null;
      Dimension prefSize = new Dimension(0,0);
      boolean fokus = false;
      int x;
      int y;

   public Picture( Image image){

      MediaTracker tracker = new MediaTracker(this);
      bild = image;
      tracker.addImage(bild,0);
      try
      {
      tracker.waitForID(0);
      }
      catch(InterruptedException exc){}
      prefSize.width = bild.getWidth(this);
      prefSize.height= bild.getHeight(this);
      addMouseListener(new MausBearbeiter());
   }

 public Dimension getPreferredSize(){
       return prefSize;
 }

 public void paint(Graphics g){
       g.drawImage(bild,0,0,this);
       g.setColor(Color.yellow);
       g.drawRect(x,y,10,10);
 }

 public boolean isFocusTraversable(){
       return true;
 }

 class MausBearbeiter implements MouseListener{
          public void mousePressed(MouseEvent e){
                fokus = true;
                x = e.getX();
                y = e.getY();
                repaint();
                ImageComp.an.requestFocus();
          }
          public void mouseExited(MouseEvent e){
                fokus = false;
```

```
		}
		public void mouseClicked(MouseEvent e){}
		public void mouseReleased(MouseEvent e){}
		public void mouseEntered(MouseEvent e){}
 }

}
```

Coding der Klasse 'CCCAlgo'

```
// CCCAlgo.java

import java.awt.*;
import java.awt.image.*;
import java.io.*;

public class CCCAlgo extends Component{

	Image bild, compressed;
	int hoehe, breite;
	int blo_h, blo_b;
	int[] pix;
	int[][] pixels;
	int summa;

	CCCAlgo(Image bild, int hoehe, int breite, int blo_h, int blo_b){

	this.bild = bild;
	this.hoehe = hoehe;
	this.breite = breite;
	this.blo_h = blo_h;
	this.blo_b = blo_b;

	first();
	second();
	third();
	fourth();
	}

	void first(){

	    pix = new int[hoehe*breite];
	 PixelGrabber pg =
	      new PixelGrabber(bild, 0, 0, breite, hoehe, pix, 0, breite);
	    boolean komplett = false;

	    try{
	    komplett = pg.grabPixels();
	    }catch (InterruptedException ign){}

	    pixels = new int[hoehe][breite];
```

```
        if(komplett){
          int k = 0;
          for(int i=0; i<hoehe; i++){
            for(int j=0; j<breite; j++){
               pixels[i][j] = pix[k];
               k++;
            }
          }
        }

    }//first


    void second(){
        double mittel;
        int grau;
        int p,q;
        int rotfora, rotforb;
        int grfora, grforb;
        int blfora, blforb;

        int arot, agruen, ablau;
        int brot, bgruen, bblau;

        int sz,zz;
        PrintWriter out = new PrintWriter(System.out, true);


     for(int i=0; i< (hoehe - ( hoehe % blo_h ) ); i += blo_h){
      for(int j=0; j< (breite - ( breite % blo_b ) ); j += blo_b){

          grau = 0;
          mittel = 0.0;
          p = 0; q = 0;
          rotfora = 0; rotforb = 0;
          grfora = 0; grforb = 0;
          blfora = 0; blforb = 0;
          arot = 0; agruen = 0; ablau = 0;
          brot = 0; bgruen = 0; bblau = 0;


        for(zz=1; zz <= blo_h; zz++){
          for(sz=1; sz <= blo_b; sz++){


             grau += (int)
                    (0.3 * (( pixels[i][j] & 0x00ff0000 ) >>>16  )+
                     0.59 * (( pixels[i][j] & 0x0000ff00 ) >>>8 )+
                     0.11 * ( pixels[i][j] & 0x000000ff ) );

           j++;
          }
          j -= (sz-1);
          i++;
        }
        i -= (zz-1);
```

```
mittel = grau/(double)(blo_h * blo_b);


for(zz=1; zz <= blo_h; zz++){
  for(sz=1; sz <= blo_b; sz++){

  int y = (int)
        (0.3 * (( pixels[i][j] & 0x00ff0000 )>>>16  )+
             0.59 * (( pixels[i][j] & 0x0000ff00 )>>>8  )+
             0.11 * ( pixels[i][j] & 0x000000ff ) );

     if( y <= mittel ){
           q += 1;
        rotfora += (( pixels[i][j] & 0x00ff0000)>>>16  );
        grfora += (( pixels[i][j] & 0x0000ff00)>>>8  );
        blfora += ( pixels[i][j] & 0x000000ff );
     }
     else{
           p += 1;
        rotforb += ((pixels[i][j] & 0x00ff0000)>>>16  );
        grforb += (( pixels[i][j] & 0x0000ff00)>>>8  );
        blforb += ( pixels[i][j] & 0x000000ff );
     }


    j++;
   }
  j -= (sz-1);
  i++;
}
i -= (zz-1);


arot = (int)( rotfora / (double)q );
agruen = (int)( grfora / (double)q );
ablau = (int)( blfora / (double)q );

brot = (int)( rotforb / (double)p );
bgruen = (int)( grforb / (double)p );
bblau = (int)( blforb / (double)p );



for(zz=1; zz <= blo_h; zz++){
  for(sz=1; sz <= blo_b; sz++){

    int y = (int)
          ( 0.3 * (( pixels[i][j] & 0x00ff0000 )>>>16  )+
               0.59 * (( pixels[i][j] & 0x0000ff00 )>>>8  )+
               0.11 * ( pixels[i][j] & 0x000000ff ));

      if ( y <= mittel )
         pixels[i][j] =
(pixels[i][j]& 0xff000000)|( arot<<16  )|( agruen<<8  )|( ablau );
```

```
                else
                   pixels[i][j] =
         (pixels[i][j]& 0xff000000)|( brot<<16  )|( bgruen<<8  )|( bblau );



            j++;
           }
          j -= (sz-1);
          i++;
         }
         i -= (zz-1);

       summa += (blo_h * blo_b + 16);

       }
      } //big fors

    } //second



    void third(){

        int k = 0;

        for(int i=0; i < hoehe; i++){
          for(int j=0; j < breite; j++){

              pix[k] = pixels[i][j];
              k++;
          }
        }
    }//third

    void fourth(){

    compressed =
         createImage(new MemoryImageSource(breite,hoehe,pix,0,breite));
    }

}//CCCAlgo
```