

1	AUFGABENSTELLUNG	2
2	LÖSUNGSANSATZ	2
2.1	ÜBERSICHT.....	2
2.2	ERKENNEN UND ISOLIEREN	2
2.2.1	<i>Schnittstelle.....</i>	2
2.2.2	<i>Algorithmus</i>	2
2.3	ROTIEREN.....	2
2.3.1	<i>Schnittstelle.....</i>	2
2.3.2	<i>Algorithmus</i>	3
2.4	PUNKTERKENNUNG	4
2.4.1	<i>Überblick.....</i>	4
2.4.2	<i>Wahl des Verfahrens.....</i>	4
2.4.3	<i>Prinzip des Punktezahlens mittels Template.....</i>	5
2.5	AUSGABE	6
2.5.1	<i>Überblick.....</i>	6
2.5.2	<i>Prinzip:</i>	6
3	UMSETZUNG.....	7
3.1	ERKENNEN UND ISOLIEREN	7
3.2	ROTIEREN.....	10
3.2.1	<i>Sollte das Bild breiter als hoch sein, wird es um 90 Grad gedreht</i>	10
3.2.2	<i>Der Rotationswinkel wird festgestellt, dabei wird folgende Idee angewandt: [BILD+Formel]</i>	10
3.3	PUNKTERKENNUNG	12
3.4	AUSGABE	13
	<i>Configuration.....</i>	13
	<i>Declaration of Variables.....</i>	13
	<i>Adding stones to the Lists</i>	13
	<i>Generates images for domino lists.....</i>	13
3.5	EINSCHRÄNKUNGEN.....	13
4	TESTS.....	14
4.1	ERKENNEN UND ISOLIEREN	14
4.2	ROTIEREN.....	14
4.3	PUNKTERKENNUNG	14
5	ZUSAMMENFASSUNG	14
5.1	AUFGABENTEILUNG:	15
6	AUFWANDSTABELLEN.....	15
7	LITERATURANGABEN.....	15
8	ANHANG.....	16
8.1	DOMINOFILTER_.JAVA.....	16
8.2	DSTONE.JAVA	24

1 Aufgabenstellung

Vorgabe ist, aus einem Bild mit beliebig vielen und beliebig platzierten Dominosteinen ein neues Bild, in dem die Steine gemäß der Spielregeln zusammengesetzt sind, zu erzeugen.

2 Lösungsansatz

2.1 Übersicht

Das Programm wurde in drei Grundbereiche aufgeteilt:

- Erkennen und Isolieren der einzelnen Steine (Andrzej Kozlowski)
- Rotieren der Steine um eine einheitliche Ausrichtung zu erreichen (Walter Jenner)
- Erkennen der Dominosteinpunkte, Spielalgorithmus und Ausgabe (Björn Schnetzinger)

2.2 Erkennen und Isolieren

2.2.1 Schnittstelle

2.2.1.1 Input

8-Bit Graustufenbild, das mehrere Dominosteine enthält (z.B.: eingescannte Dominosteine).

2.2.1.2 Output

Eine LinkedList, die DStone Objekte beinhaltet. Die Dstone Objekte werden mit den einzelnen freigestellten Dominosteinen (als ImagePlus) und den zugehörigen Binärbildern (ebenfalls als ImagePlus) initialisiert.

2.2.2 Algorithmus

Das Grauwertbild wird zuerst in ein Binärbild umgewandelt. Der zweite Schritt ist die Regionenfindung; hier wird jeder einzelne Dominostein mit einem Label versehen. Als Nächstes wird die Bounding Box der einzelnen Steine mit Hilfe der Labels ermittelt, freigestellt und in eine Liste übergeben.

2.3 Rotieren

2.3.1 Schnittstelle

2.3.1.1 Input

Freigestelltes Graustufenbild in dem genau 1 Dominostein ist. Eventuelle Ecken von anderen Steinen sind in diesem Bild auch vorhanden.

Freigestelltes Binärbild in dem auch genau 1 Dominostein ist, allerdings wurden hier eventuelle Ecken von anderen Steinen schon entfernt.

2.3.1.2 Output

Rotiertes, freigestelltes Graustufenbild in der Größe des Dominosteins.

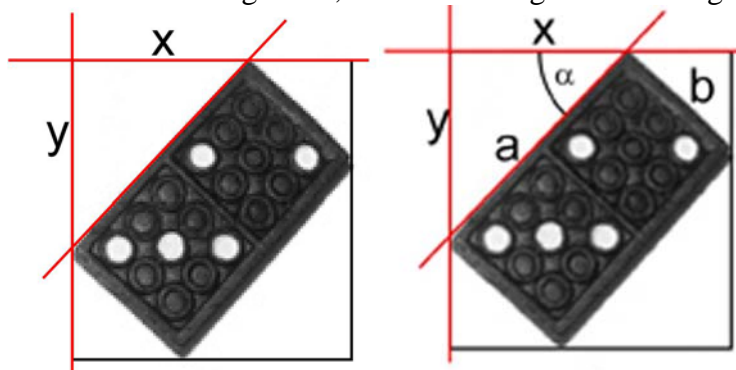
Rotiertes, freigestelltes Binärbild in der Größe des Dominosteins.

2.3.2 Algorithmus

Das Binärbild dient dazu um den benötigten Rotationswinkel und auch die „region of interest“ (ROI), die für das Freistellen benötigt wird, zu ermitteln. Mit den bekommenen Daten wird dann sowohl das Graustufenbild als auch das Binärbild gedreht und freigestellt.

2.3.2.1 Ablauf

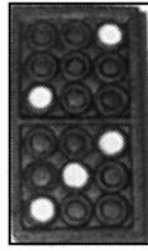
1. Breite und Höhe des Inputs wird ermittelt.
2. Sollte das Bild breiter als hoch sein, wird es um 90 Grad gedreht.
3. Der Rotationswinkel wird festgestellt, dabei wird folgende Idee angewandt:



4. Das Bild wird dem Rotationswinkel entsprechend so gedreht, dass es die längere Seite des Steines vertikal ist.



5. Nun wird das Bild noch so freigestellt, dass nur noch der Dominostein übrig bleibt.



6. Schlussendlich wird die Rotation und das Freistellen auch noch auf das Graustufenbild angewandt, das bis jetzt unverändert blieb.

2.4 Punkterkennung

2.4.1 Überblick

Anforderung: Nach dem die Dominosteine richtig rotiert und zugeschnitten worden sind, müssen noch ihre Werte, also die Anzahl der Punkte an den 2 Seiten, erkannt und zugewiesen werden.

2.4.2 Wahl des Verfahrens

Es standen 2 mögliche Techniken zur Wahl: entweder man sucht über Regionserkennung (mit einem Floodfill Algorithmus) die weißen Flächen im Bild und zählt diese, oder man verwendet eine Template. Mit der Template werden alle möglichen Positionen der weißen Punkte eines Dominosteins überprüft, mit dem Ergebnis könne man dann den richtigen Wert des Steins feststellen.

Wir entschieden uns für die 2. Möglichkeit, denn die Template-Lösung schien einige eindeutige Vorteile gegenüber der Regionserkennung zu haben:

- + Geringer Rechenaufwand und somit gute Performance auch bei mehreren Steinen
- + aufgrund des Aufbaues eines Dominosteins, würde sich eine Template hervorragend eignen
- + gute Strukturierbarkeit (Programm - Komplexität würde sich in Grenzen halten)
- + Störungen in den Randbereichen des Bildes würden das Endergebnis nicht beeinträchtigen

2.4.3 Prinzip des Punktezählens mittels Template

Aufgrund des bestimmten Aufbaues eines Dominosteins geht hervor, dass jede Seite genau 9 Positionen hat, an denen eventuell ein weißer Punkt vorhanden sein kann. Es ist nicht möglich, dass ein weißer Punkt zwischen dem Muster auftritt. (Abb. 1, mittleres Bild). Die gegenüberliegende Seite des Steines weist, dasselbe Muster auf, und kann deshalb gleich behandelt werden. (Abb. 1, letztes Bild).

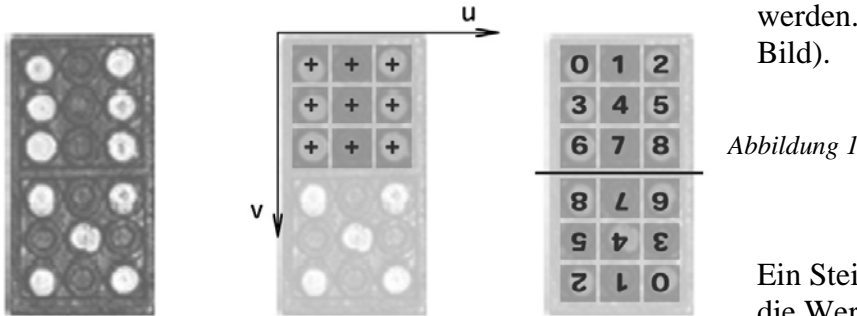


Abbildung 1

Ein Stein kann an einer Seite die Werte 0 bis 6 enthalten.

Abbildung 2 zeigt die

möglichen Kombinationen für die Werte 1 bis 6. Für Wert 2 und 3 gibt es weiters 2 Möglichkeiten, wie der Wert aussehen kann.



Abbildung 2



Da sich die Template auf bestimmte Koordinaten bezieht, ist sie nur auf eine abgestimmte Bildgröße anwendbar. Das kann aber leicht durch ein richtiges Skalieren erreicht werden.



Bei der Punkteerkennung muss eine gewisse Fehlertoleranz berücksichtigt werden. Die Steine können durch das vorherige Rotieren leicht verschoben bzw. noch leicht schief liegen. Obwohl die Template von Haus aus eine gewisse Toleranz zulässt, weil sie sich immer auf den Mittelpunkt eines idealen weißen Punktes bezieht, reicht das alleine noch nicht aus, um Abweichungen auszugleichen. Hauptproblem ist, dass die weißen Punkte eines Steines zu klein sind. Sie müssen vergrößert werden, bevor sie mit der Template abgefragt werden können.



Hier kommt das Erodieren ins Spiel. Erodieren vergrößert weiße Stellen eines Binärbildes. Mehrmaliges Erodieren eines Dominosteinbildes schafft alle notwendigen Voraussetzungen. Abb. 03 zeigt die Ausgangsbilder, die bereits die richtige Größe haben. Abb. 04 zeigt die Bilder nach dem Erodieren und die Kreuze der Koordinaten, die in der Template eingetragen sind.



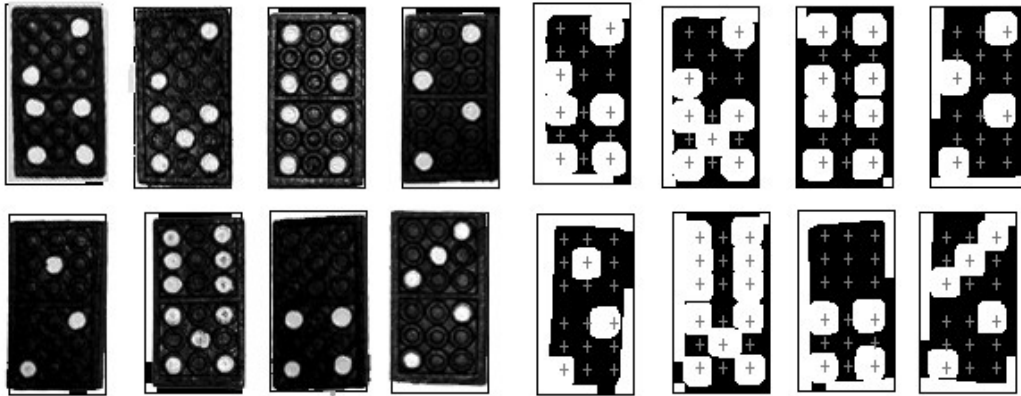


Abbildung 3

Abbildung 4

2.5 Ausgabe

2.5.1 Überblick

Wählt den 1. Stein der Liste aus und hängt die übrigen Dominosteine daran an (entweder links, oder rechts), sofern sie auch passen. Die aneinander gereihten Dominosteine werden als Lösungsbild ausgegeben.

Nicht passende Steine werden im Ausschussbild, das zusätzlich die jeweiligen Werte der Dominosteine darstellt, ausgegeben.

2.5.2 Prinzip:

Es werden weitere 2 Listen, die Dominostein-Objekte enthalten (DStones) verwendet. Zur Quell Liste, wird eine Liste erzeugt, die alle Lösungselemente enthält und auch eine, die die Ausschusselemente enthält.

Erstellung der Lösungsliste, die alle passenden Steine enthält:

Jeder Dominostein besitzt für jede seiner 2 Seiten einen Wert. Anhand von diesem Wird entweder rechts oder links ein anderer Stein angehängt. Ein Dominostein kann auch um 180 Grad gedreht werden und somit mit jeder Seite links oder rechts an die Liste angehängt werden. Das muss auch bei der späteren Ausgabe bei den Ergebnisbildern berücksichtigt werden.

Abbildung 6

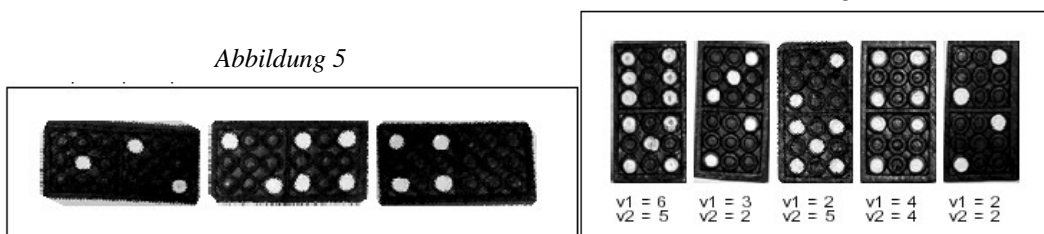
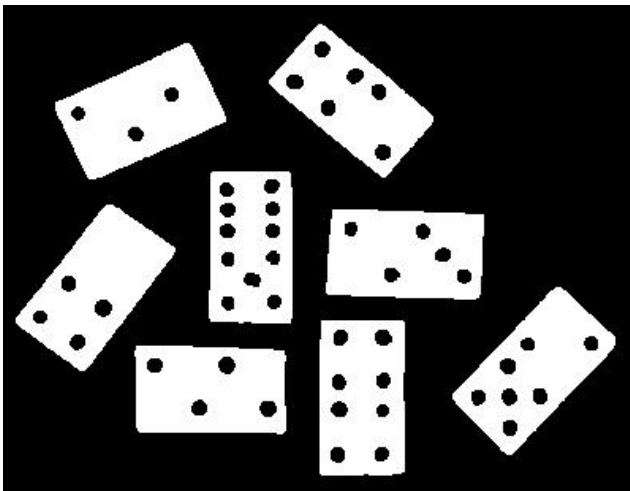


Abbildung 5

3 Umsetzung

3.1 Erkennen und Isolieren

Das Graustufenbild wird in ein Binärbild umgewandelt, ein Closing durchgeführt, um kleine Fehler zu entfernen, invertiert, und wieder ein Closing durchgeführt, um noch mehr Fehler zu entfernen. Dafür wurden die schon in der Klasse ImageProcessor implementierten Methoden verwendet.



Der nächste Schritt ist es, jeden Dominostein mit einem Label zu versehen. Da die Annahme gemacht wird, dass jedes Hintergrundpixel

den Wert bzw. Label 0 hat und jedes Vordergrundpixel den Wert/Label 1, müssen zuerst die weißen Pixel auf den Wert 1 gesetzt werden. Das geschieht mit der einfachen Methode `change255to1()`.

Weiters werden die Labels mittels der Breadth-first-Variante des iterativen Flood-Fill-Algorithmus gesetzt (Algorithmus und Code entnommen aus Burger/Burge – Digitale Bildverarbeitung Kap. 11). Dies geschieht im Folgenden Codeteil, in der die Methode `floodFill()` aufgerufen wird:

```

for(int v = 0; v < ws.getHeight(); v++)
{
    for(int u = 0; u < ws.getWidth(); u++)
    {
        if(ws.getPixel(u,v) == 1)
        {
            floodFill(ws,u,v,m);
            m++;
        }
    }
}

```

Sind die Labels gesetzt, wird ein Array vom Typ BoundingBox (selbst implementiert – enthält die kleinsten x, y und die größten x, y Koordinaten) angelegt, das genauso groß ist, wie die Anzahl der Labels. Mithilfe dieser Bounding Boxes sollen die einzelnen Steine freigestellt werden. Da die Labels hierbei eine wichtige Rolle spielen, wird festgelegt, dass der Index + 2 jedes Arrayelementes der Nummer des Labels entspricht (die Labelnummerierung beginnt mit 2). Aufgrund des weiter unten beschriebenen Suchalgorithmus wird zunächst die kleinste x und die kleinste y Koordinate größer als die Breite bzw. Höhe des Bildes gewählt und die größte x und die größte y Koordinate gleich 0 gesetzt. Nun wird das ganze Bild durchlaufen und überprüft, ob das aktuelle Pixel (Koordinaten u,v) ein Label hat (also größer 0). Falls ja, werden die einzelnen Koordinaten der Boundingboxes überprüft (der Index der gerade überprüften BoundingBox entspricht dem Pixelwert an der Stelle u,v minus 2). Ist die kleinste x-Koordinate größer als die aktuelle u-Koordinate, dann bekommt diese den Wert von u. Ist die größte x-Koordinate kleiner als die aktuelle u-Koordinate, dann bekommt diese den Wert von u. Es wird analog mit der kleinsten und größten y-Koordinate in Bezug auf v verfahren.

```

for(int v = 0; v < ws.getHeight(); v++)
{
    for(int u = 0; u < ws.getWidth(); u++)
    {
        if(ws.getPixel(u,v) > 0)
        {
            if(boxes[ws.getPixel(u,v)-2].getX1() > u)
            {
                boxes[ws.getPixel(u,v)-2].setX1(u);
            }
            if(boxes[ws.getPixel(u,v)-2].getX2() < u)
            {
                boxes[ws.getPixel(u,v)-2].setX2(u);
            }
            if(boxes[ws.getPixel(u,v)-2].getY1() > v)
            {
                boxes[ws.getPixel(u,v)-2].setY1(v);
            }
            if(boxes[ws.getPixel(u,v)-2].getY2() < v)
            {
                boxes[ws.getPixel(u,v)-2].setY2(v);
            }
        }
    }
}

```

Als nächstes werden anhand der BoundingBoxes in einer Schleife die Regions of Interest gesetzt und mit neuen ImageProcessors die Dominosteine aus dem Grauwertbild sowie dem Binärbild freigestellt und ImagePlus Objekte aus diesen ImageProcessors angelegt. Ist das geschehen, werden eventuelle Ecken und Teile anderer Dominosteine, die möglicherweise mitgeschnitten worden sind, mit der Methode `cleanseBinary()`

aufgerufen; in dieser wird jedes Pixel, das nicht den aktuellen Labelwert oder den Wert des Hintergrunds hat, auf den Hintergrundwert gesetzt.

Weiters werden die ImageProcessors dafür verwendet, um die Dstone Objekte zu initialisieren, die dann in eine LinkedList geladen werden.

```
LinkedList dominos = new LinkedList();

for(int j = 0; j < boxes.length; j++)
{
    ip.setRoi(boxes[j].getX1(),boxes[j].getY1(),
             boxes[j].getX2()-boxes[j].getX1(),
             boxes[j].getY2()-boxes[j].getY1());
    ws.setRoi(boxes[j].getX1(),boxes[j].getY1(),
             boxes[j].getX2()-boxes[j].getX1(),
             boxes[j].getY2()-boxes[j].getY1());
    ImageProcessor currentStone = ip.createProcessor(boxes[j].getX2()-
    boxes[j].getX1(),
    boxes[j].getY2()-
    boxes[j].getY1());

    ImageProcessor currentBinary = ws.createProcessor(boxes[j].getX2()-
    boxes[j].getX1(),
    boxes[j].getY2()-
    boxes[j].getY1());
    currentStone = ip.crop();
    currentBinary = ws.crop();
    cleanseBinary(currentBinary,j+2);
    String title = "Stone " + (j+1);
    ImagePlus img = new ImagePlus(title,currentStone);
    ImagePlus binary = new ImagePlus(title + " binary",currentBinary);
    dominos.add(new DStone(img,binary));
}
```

Bis auf die Methoden, die zur sauberen Konvertierung in ein Binärbild benutzt werden, werden die anderen Methoden in der Methode `isolateStones()` aufgerufen; die Hierarchie ist also folgende:

```
isolateStones()
{
    change255to1()
    floodFill()
    Die Ermittlung der Bounding Boxes
    Erstellung der ImageProcessors und -Pluses
    cleanseBinary()
    Rückgabe einer LinkedList mit fertigen Dstones
}
```

3.2 Rotieren

3.2.1 Sollte das Bild breiter als hoch sein, wird es um 90 Grad gedreht

```
if (width > height){
    //rotate binary image 90 degrees
    ImageProcessor ipTemp= ipBinDup.rotateLeft();
    this.binary = new ImagePlus("Binary Stone Image",ipTemp);
    ipBinDup= this.binary.getProcessor();

    //rotate original image too
    ImageProcessor ipTemp2= ipOrigDup.rotateLeft();
    this.img = new ImagePlus("Stone Image",ipTemp2);
    ipOrigDup= this.img.getProcessor();

    //reset width and height
    width= ipBinDup.getWidth();
    height= ipBinDup.getHeight();
}
```

Sollte das Bild breiter als hoch sein, wird es „aufgestellt“. Dazu verwenden wir die von IJ zur Verfügung gestellte Funktion ImageProcessor ImageProcessor::rotateLeft() die das Bild um 90° gegen den Uhrzeigersinn dreht. Damit das Ganze auch funktioniert, muss ich einen temporären ImageProcessor anlegen.

Die Rotation wird auf beide Dominobilder (s/w und Graustufen) angewandt.

Zum Schluss werden die beiden Variablen width und height noch aktualisiert, da die auch für den weiteren Verlauf benötigt werden.

3.2.2 Der Rotationswinkel wird festgestellt, dabei wird folgende Idee angewandt: [BILD+Formel]

```
for (int i= 0; i<width; i++){
    if (ipBinDup.getPixel(i,0)==0){
        x= i;
        break;
    }
}
for (int j= 0; j<height; j++){
    if (ipBinDup.getPixel(0,j)==0){
        y= j;
        break;
    }
}
```

X wird ermittelt indem man den ersten schwarzen Pixel von links in der ersten Zeile sucht, und Y indem man den ersten schwarzen Pixel von oben in der ersten Spalte sucht. Wir gehen davon aus, dass das Bild so freigestellt ist, dass in der ersten Zeile/Spalte immer ein schwarzer Pixel ist, weil wir das im vorhergehenden Programmteil so implementiert haben.

Mit X und Y kann man dann die Winkelberechnung mittels Tangens vornehmen:

```

alpha= java.lang.Math.toDegrees(java.lang.Math.atan2(y,x));
a= Math.sqrt(Math.pow(x,2)+Math.pow(y,2));
b= Math.sqrt(Math.pow(width-x,2)+Math.pow(height-y,2));

```

alpha ist der Winkel, und a und b entsprechen den ungefähren Seitenlängen des Dominosteins.

```

if (a>b){
    alpha= alpha+ 90;
}

```

Sollte a größer als b sein, also der Winkel bzgl. der langen Seite errechnet worden sein, drehe ich den Stein um 90° mehr, damit der Stein wieder aufrecht steht.

Generell wird das Bild nur gedreht wenn x und y größer 4 sind. Damit werden Steine, die von vornherein schon richtig gedreht sind, nicht noch einmal rotiert werden.

Danach wird die Rotation mit der von IJ vorimplementierten Funktion void ImageProcessor::rotate(int angle) gedreht.

Mit der Funktion Roi getCropRoi() wird die ROI herausgefunden, mit der man das Bild noch zurechtschneiden kann, damit nur noch der Dominostein noch im Bild ist. Mit ImageProcessor ImageProcessor::crop() wird der Dominostein dann mit dem gesetzten ROI freigestellt.

```

ipBinDup.rotate(alpha);
this.binary= new ImagePlus("Binary Image",ipBinDup);

Roi roi= getCropRoi();

ipBinDup.setRoi(roi);
ImageProcessor ip2 = ipBinDup.crop();
this.binary = new ImagePlus("Binary Stone Image",ip2);

ipOrigDup.rotate(alpha);
ipOrigDup.setRoi(roi);
ImageProcessor ip3 = ipOrigDup.crop();

this.img = new ImagePlus("Stone Image",ip3);

```

Noch kurz zur Methode die den ROI findet, mit dem der Stein freigestellt wird. Ich brauche die niedrigsten und die höchsten x und y Werte im Bild, damit ich die Bounding Box um den Dominostein definieren kann. Dazu reichen 2 Schleifen, die über das Bild laufen:

```

int xIn= ipBin.getWidth(), xOut= 0, yIn= height, yOut= 0;

for (int y= 0; y< height; y++){
    for (int x= 0; x< width; x++){
        if (ipBin.getPixel(x,y) == 0){
            if (x < xIn){
                xIn = x;
            }
        }
    }
}

```

```

        if (x > xOut){
            xOut = x;
        }
        if (y < yIn){
            yIn = y;
        }
        if (y > yOut){
            yOut = y;
        }
    }
}

```

3.3 Punkterkennung

Die Teilaufgabe „Weiße Punkte Erkennung der Dominosteine“ wurde folgendermaßen strukturiert:

```
void setPoints(boolean showTemplateCoordinates):
```

Dies ist die Hauptmethode. In ihr wurde die Template definiert, die Größe des Bildes auf das sich die Template bezieht und die Anzahl der Erodier Durchgänge. Sie greift auf das Binärbild des Dominosteins zu, erzeugt eine Kopie für die Skalierung und für das Erodieren. Auf jede Seite des Steins wird die Methode checkPoints angewendet. Diese generiert einen Array mit Flags, der wiederum an die Methode findSetValues übergeben wird. Das Wechseln der Seite des Steins wird durch Flippen des Bildes erreicht. Optional ist die Ausgabe der bearbeiteten Bilder, auf denen die Template Koordinaten mit Fadenkreuze eingezeichnet werden.

```
ImageProcessor showTemplateCoordinates(ImageProcessor ip, int[][] t):
```

Ursprünglich nur für Debugzwecke geschrieben, zeigt diese Methode aber sehr gut die Koordinaten, der Template, welche überprüft werden.

```
boolean checkPoints(ImageProcessor ip, int[][] t):
```

Diese Methode überprüft alle Punkte in der Template und erzeugt das Punktmuster (von Abb.2)

Sie generiert einen booleschen Array und überprüft jede Koordinate, die in der Template eingetragen ist, mit dem Bildinhalt. Je nachdem, ob sie ein weißes Pixel (= weißer Punkt) oder schwarzes Pixel (= kein weißer Punkt) findet, setzt sie das Flag oder nicht. Schlussendlich gibt sie den booleschen Array zurück.

```
private void findSetValues(boolean[] tf):
```

Diese Methode identifiziert das von checkpoints erstellte Muster. Sie überprüft die in Abb. 2 eingezeichneten weißen oder schwarzen Mussfelder. Sie ruft die Methode setValue auf.

```
private void setValue(int v):
```

Je nachdem, ob schon der Punktwert der 1. Hälfte des Bildes oder nicht erkannt wurde, schreibt sie den Wert für den aktuellen Dominostein.

3.4 Ausgabe

Anhängen der Steine und Ausgabe wird zwecks Einfachheit in einer Methode erreicht. Methode ist in 4 wichtige Unterbereiche zusammengefasst: void sortStones(LinkedList source):

Configuration

legt die Abstände zwischen den Steinen in den Ausgabebildern fest.

Declaration of Variables

Enthält alle notwendigen Methodenvariablen für das Sortieren und für die Ausgabe

Adding stones to the Lists

beginnt mit einem Stein (erstem Stein der Quellliste). Die anderen Steine werden entsprechend ihrer Wertigkeit überprüft, ob sie an die Ursprungskette passen und angehängt. Die anderen Steine werden in eigener trash Liste gesammelt. Orientation der Steine in der Lösungsliste werden in den einzelnen Steinobjekten festgelegt.

Generates images for domino lists

Anhand der Anzahl der Lösungsliste (sol) und der Ausschussliste (trash) werden 2 Bilder erzeugt, in welches die einzelnen Steine eingefügt werden, wobei aber zuvor festgelegte Abstände eingehalten werden. Bei dem Lösungsbild wird die Orientation der Steine ebenfalls berücksichtigt. Im trash-Bild werden die Steine aufrecht stehend mit den jeweiligen Punktwerten ausgegeben.

3.5 Einschränkungen

- Dominosteine dürfen sich nicht überlappen
- Alle Dominosteine befinden sich zur Gänze im Bild
- Mindestens ein Dominostein ist im Bild enthalten
- nur Graustufenbilder werden unterstützt

- der Hintergrund selbst sollte keine zu großen Helligkeitsunterschiede aufweisen, da sonst Teile als Region erkannt werden könnten.
- Dominosteine dürfen nicht Verzerrt werden (sollen rechteckig sein).

4 Tests

4.1 Erkennen und Isolieren

Wenn der Hintergrund zu große Intensitätsunterschiede aufweist, werden Teile davon als Region erkannt, da die Umwandlung in ein Binärbild nicht reibungslos abläuft. Auch wenn kleine Fehler, die das Closing nicht entfernt hat, auftreten, verursachen eine ungewollte Regionenerkennung.

4.2 Rotieren

Nach den ersten Implementierung und den darauffolgenden Tests stellte sich heraus, dass wir noch nicht ganz am Ziel waren. Die erste Umsetzung bediente sich noch dem freigestellten Graustufenbild, bei dem u.U. noch Ecken von anderen Dominosteinen im Randbereich zu finden waren. Das war beim Zurechtschneiden der rotierten Steine immer ein Problem, und auch beim Feststellen des Rotationswinkels konnte es ein Problem sein, und zwar dann wenn sich der Rest eines anderen Dominosteines genau im linken oberen Eck des Bildes befand. Also mussten wir uns etwas überlegen – die einfachste und gleichzeitig beste Lösung war, beim Labeling gleich eine von anderen Dominosteinen bereinigte Version als Binärbild zu speichern., welches in weiterer Folge als Berechnungsgrundlage des Rotationswinkels und des Freistellungsrahmens diente. Die daraus gewonnen Ergebnisse wurden dann auch auf das – noch unsaubere – Graustufenbild angewandt, wodurch wir makellose Ergebnisse bekamen.

4.3 Punkterkennung

Die Weiße Punkte Erkennung wurde mit vielseitigen Bildern getestet. Bei normaler Scanqualität schien sie nie Schwierigkeiten zu haben. Probleme tauchen nur auf, wenn die Dominosteine schlecht abfotografiert wurden und es zu geometrischen Verzerrungen der Bilder gekommen ist. Dieser Fall ist aber verschmerzbar.

5 Zusammenfassung

Das Ziel wurde erreicht, mit allen von uns verwendeten Testbildern wurde ein zufrieden stellendes Ergebnis erreicht.

Große Probleme gab es beim Labelling, da wir zuerst die Sequentielle Regionenfindung anwenden wollten. Da die Implementierung dieser nicht geglückt ist, wählten wir die Breadth-First Variante des Floodfill Algorithmus, die tadellos funktioniert.

Unsicherheit gab es bei der Punkterkennung, ob sie mittels Templates auch in der Praxis funktionieren würde. Schließlich aber stellte sich heraus, dass unsere Bedenken unbegründet waren.

5.1 Aufgabenteilung:

Walter Jenner: Rotieren der Steine um eine einheitliche Ausrichtung zu erreichen

Andrzej Kozlowski: Erkennen und Isolieren der einzelnen Steine

Björn Schnetzinger: Erkennen der Dominosteinpunkte, Spielalgorithmus und Ausgabe

6 Aufwandstabellen

	Konzept, Implementierung	Finetuning, Fehlerbehebung	Dokumentation
Jenner Walter	3xFluchen	2xHusten	1x“Draufschießen“
Andrzej Kozlowski	1 Mond, 1 Sonnenaufgang, 1xOben-Ohne	1 Broteinheit Schokolade, 5 Pixel, 100 Lungentorpedos	Krummer Rücken, 1 zerbrochenes Schwert
Björn Schnetzinger	$123^{\wedge} 345$ gespaltene Elektronen, 3 tote Orks, -10 Charisma	$10^{\wedge} 75$ gewonnene Gehirnzellen, geprellte Hand, +2 Stärke	50 Haare mit Fett, 1 Flasche Shampoo +346 Erfahrungspunkte

	Konzept, Implementierung	Finetuning, Fehlerbehebung	Dokumentation
Jenner Walter	12h	8h	7h
Andrzej Kozlowski	14h	11h	7h
Björn Schnetzinger	15h	9h	8h

7 Literaturangaben

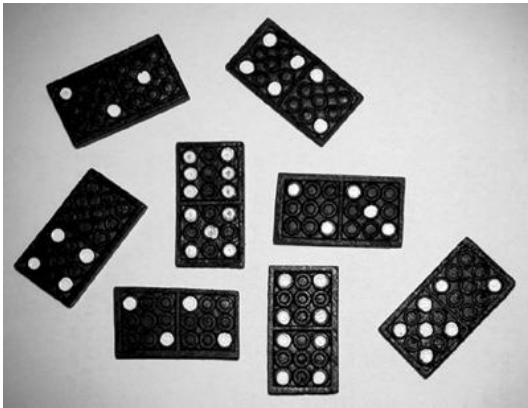
Digitale Bildverarbeitung – Eine Einführung mit Java und ImageJ

Burger/Burge

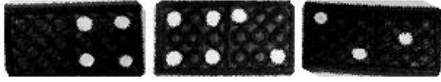
Springer Verlag

ISBN: 3-540-21465-8

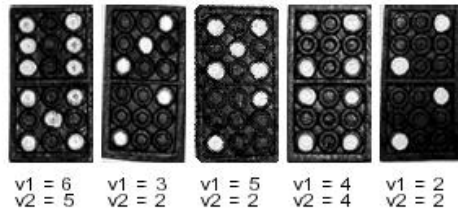
8 Beispielbilder



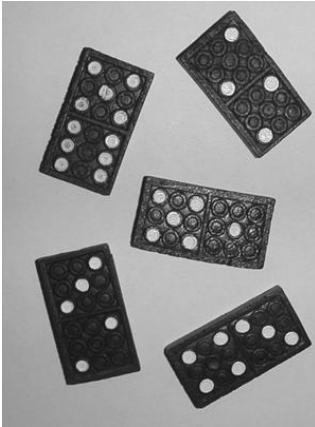
Ausgangsbild 1



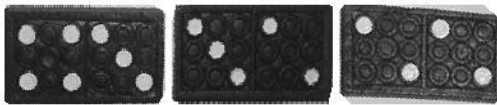
Endbild 1 – Solution



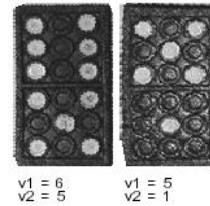
Endbild 1 - Trash



Ausgangsbild 2



Endbild 2 – Solution



Endbild 2 – Trash

9 Anhang

9.1 DominoFilter_.java

```
import ij.*;
import ij.gui.*; //nicht vergessen!
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import java.awt.*;
import java.util.LinkedList;
import java.util.Iterator;
import ij.gui.NewImage;

/**
 * This ImageJ plugin detects domino stones in a grayscale picture, counts their
 * points and creates a new image with the solution of the game.
 *
 * @author Walter Jenner, Andrzej Kozlowski, Bjoern Schnetzinger
 * @since 05/2005
 */
```

```

* @version 1.0
*/
public class DominoFilter_ implements PlugInFilter
{
    int m = 2;
    int min_m = 2;
    boolean debug = false;

    public int setup(String arg, ImagePlus ip)
    {
        if (arg.equals("about"))
            {showAbout(); return DONE;}
        return DOES_8G;
    }

    public void run(ImageProcessor ip)
    {
        ImageProcessor temp = ip.duplicate();
        temp.autoThreshold();
        temp.dilate();
        temp.erode();
        temp.invert();
        temp.dilate();
        temp.erode();
        LinkedList dominos = isolateStones(ip,temp);

        Iterator iter= dominos.iterator();

        // rotate all domino images of the list
        while (iter.hasNext()){
            DStone s= (DStone)iter.next();
            s.rotate();
            //s.getImg().show();
        }

        if (debug) IJ.write(" -----PROGRAMM STARTED AGAIN -----
        ----- ");

        // setPoints for all domino stones of the list
        Iterator iter2= dominos.iterator();
        while (iter2.hasNext()) {
            ((DStone)iter2.next()).setPoints(false);
        }

        // simulation of the domino game: shows 2 images: solution and trash
        sortStones(dominos);
    }

    void showAbout()
    {
        IJ.showMessage("About this plugin...",
            "Detects domino stones and solves the game."
        );
    }

    /**
     * This method determines the location of domino stones, draws bounding boxes
     * around them and crops them into new <code>ImagePlus</code> objects. Then it
     creates <code>DStone</code>
     * objects with these <code>ImagePlus</code> objects and puts them into a
     <code>LinkedList</code>.
     *
     * @author Andrzej Kozlowski
     *
     * @param ip the original 8-bit grayscale picture, used for final cropping
     * @param ws the binarized grayscale picture, used for most operations like region
     labelling
     * and determining the bounding boxes
     * @return a <code>LinkedList</code> with <code>DStone</code> objects that contain
     the cropped domino images
     */
}

```

```

LinkedList isolateStones(ImageProcessor ip, ImageProcessor ws)
{
    change255to1(ws);
    for(int v = 0; v < ws.getHeight(); v++)
    {
        for(int u = 0; u < ws.getWidth(); u++)
        {
            if(ws.getPixel(u,v) == 1)
            {
                floodFill(ws,u,v,m);
                m++;
            }
        }
    }

    BoundingBox[] boxes = new BoundingBox[m-min_m];

    for(int i = 0; i < boxes.length; i++)
    {
        boxes[i] = new BoundingBox(ws.getWidth()+1,ws.getHeight()+1);
    }

    for(int v = 0; v < ws.getHeight(); v++)
    {
        for(int u = 0; u < ws.getWidth(); u++)
        {
            if(ws.getPixel(u,v) > 0)
            {
                if(boxes[ws.getPixel(u,v)-2].getX1() > u)
                {
                    boxes[ws.getPixel(u,v)-2].setX1(u);
                }
                if(boxes[ws.getPixel(u,v)-2].getX2() < u)
                {
                    boxes[ws.getPixel(u,v)-2].setX2(u);
                }
                if(boxes[ws.getPixel(u,v)-2].getY1() > v)
                {
                    boxes[ws.getPixel(u,v)-2].setY1(v);
                }
                if(boxes[ws.getPixel(u,v)-2].getY2() < v)
                {
                    boxes[ws.getPixel(u,v)-2].setY2(v);
                }
            }
        }
    }

    LinkedList dominos = new LinkedList();

    for(int j = 0; j < boxes.length; j++)
    {
        ip.setRoi(boxes[j].getX1(),boxes[j].getY1(),
                boxes[j].getX2()-boxes[j].getX1(),
                boxes[j].getY2()-boxes[j].getY1());
        ws.setRoi(boxes[j].getX1(),boxes[j].getY1(),
                boxes[j].getX2()-boxes[j].getX1(),
                boxes[j].getY2()-boxes[j].getY1());
        ImageProcessor currentStone = ip.createProcessor(boxes[j].getX2()-
                boxes[j].getX1(),
                boxes[j].getY2()-
                boxes[j].getY1());

        ImageProcessor currentBinary = ws.createProcessor(boxes[j].getX2()-
                boxes[j].getX1(),
                boxes[j].getY2()-

```

```

        boxes[j].getY1());
        currentStone = ip.crop();
        currentBinary = ws.crop();
        cleanseBinary(currentBinary, j+2);
        String title = "Stone " + (j+1);
        ImagePlus img = new ImagePlus(title, currentStone);
        ImagePlus binary = new ImagePlus(title + " binary", currentBinary);
        //img.show();
        //binary.show();
        dominos.add(new DStone(img, binary));
    }

    return dominos;
}

/**
 * Searches for all pixels that are not background pixels (value <code>0</code>)
 * nor labelled (value <code>label</code>)
 * and changes them into background pixels. This method is used to facilitate the
rotation of
 * the domino images.
 *
 * @author Andrzej Kozlowski
 *
 * @param ip the binary image of each domino stone
 * @param label the label value that is to be preserved
 */
void cleanseBinary(ImageProcessor ip, int label)
{
    for(int u = 0; u < ip.getWidth(); u++)
    {
        for(int v = 0; v < ip.getHeight(); v++)
        {
            if(ip.getPixel(u,v) != label && ip.getPixel(u,v) != 0)
            {
                ip.putPixel(u,v,0);
            }
            if(ip.getPixel(u,v) == label)
            {
                ip.putPixel(u,v,255);
            }
        }
    }

    ip.invert();
}

/**
 * Changes all pixels that have value <code>255</code> to value <code>1</code>
 (used for region labelling).
 *
 * @author Andrzej Kozlowski
 *
 * @param ip the binary image with all domino stones
 */
void change255to1(ImageProcessor ip)
{
    for(int u = 0; u < ip.getWidth(); u++)
    {
        for(int v = 0; v < ip.getHeight(); v++)
        {
            if(ip.getPixel(u,v) == 255)
            {
                ip.putPixel(u,v,1);
            }
        }
    }
}

/**

```

```

* Labels each region with a unique value (first value is <code>min_m</code>).
*
* @author Andrzej Kozlowski
*
* @param ip the binary image with all domino stones
* @param x the x-coordinate of the first pixel, that is not a background or
already labelled pixel
* @param y the y-coordinate of the first pixel, that is not a background or
already labelled pixel
* @param label the value with which the current pixel is to be labelled, if not
background
* or already labelled pixel
*/
void floodFill(ImageProcessor ip, int x, int y, int label)
{
    LinkedList q = new LinkedList();
    q.addFirst(new Node(x,y));
    while(!q.isEmpty())
    {
        Node n = (Node) q.removeLast();
        if((n.x>=0) && (n.x<ip.getWidth()) && (n.y>=0) &&
(n.y<ip.getHeight()) &&
            ip.getPixel(n.x,n.y) == 1)
        {
            ip.putPixel(n.x,n.y,label);
            q.addFirst(new Node(n.x+1,n.y));
            q.addFirst(new Node(n.x,n.y+1));
            q.addFirst(new Node(n.x,n.y-1));
            q.addFirst(new Node(n.x-1,n.y));
        }
    }
}

/**
* Chooses the first stone and tries to append other stones (with domino stone
rules).
* It shows 2 output images: First, the "solution", for the suitable stones. A
second,
* called "trash", for all not suitable stones.
*
* @author Björn Schnetzinger
*
* @param source that contains the already rotated and cropped domino stone
images.
*/
public void sortStones(LinkedList source)
{
    // =====
    // CONFIGURATION:
    // =====

    // Output image for solution:
    int offset_x = 5; // offset between all images
    int border = 20; // offset around the images.
    int textheight = 20;

    // Output size of an image on the output images:
    int W_IDEAL = ((DStone)source.getFirst()).getImg().getHeight();
    int H_IDEAL = ((DStone)source.getFirst()).getImg().getWidth();
    //int W_IDEAL = 100;
    //int H_IDEAL = 53;

    // =====
    // DECLARATION OF VARIABLES:
    // =====

    LinkedList sol = new LinkedList(); // solution: will contain all suitable
stones

    int sol_start = 0; // first value of the solution
    int sol_end = 0; // last value of the solution

```

```

stones
LinkedList trash = new LinkedList(); // trash: contains non suitable

DStone s;

int n_lsg; // number of domino stones of the solution list
int n_trash; // number of domino stones of the trash list

ImagePlus result; // solution output image
ImageProcessor result_ip;

ImagePlus resultTrash; // trash output image
ImageProcessor resultTrash_ip;

// =====
// ADDING STONES TO THE LISTS:
// =====

/* Starting with the first stone, adding it to the solution list */
sol.add(source.getFirst());
source.removeFirst(); // important, that the first stone isn't used a
second time later!
sol_start = ((DStone)sol.getFirst()).getV1();
sol_end = ((DStone)sol.getFirst()).getV2();

solution
the
/* adds all suitable stones of the LinkedList source to the LinkedList
* (sol): left or right. Otherwise if they do not suit, they are added to

* LinkedList trash: */
Iterator i = source.iterator();
while (i.hasNext()) {
s = (DStone)i.next();
if (debug) {
IJ.write(" ");
IJ.write("take stone.");
IJ.write("LSG: start=" + sol_start + ", end=" + sol_end);
}

/* appends the chosen stone to the start of the solution list: */
if (s.getV2() == sol_start) {
s.setOrientation(true);
sol.addFirst(s);
sol_start = s.getV1();
if (debug) IJ.write("stone added to lsg LEFT");
/* appends the chosen stone to the end of the solution list: */
} else if (s.getV1() == sol_end) {
s.setOrientation(true);
sol.addLast(s);
sol_end = s.getV2();
if (debug) IJ.write("stone added to lsg RIGHT");
/* appends the chosen stone to the start of the solution list.
* Its orientation must be 180 degree rotatet: */
} else if (s.getV1() == sol_start) {
s.setOrientation(false);
sol.addFirst(s);
sol_start = s.getV2();
if (debug) IJ.write("stone added to lsg LEFT");
/* appends the chosen stone to the end of the solution list.
* Its orientation must be 180 degree rotatet: */
} else if (s.getV2() == sol_end) {
s.setOrientation(false);
sol.addLast(s);
sol_end = s.getV1();
if (debug) IJ.write("stone added to lsg RIGHT");

/* if a stone does match the first or last value of the solution
list,
* it is added to the trash list: */
} else {
trash.add(s);

```

```

        if (debug) IJ.write("stone does not match, added to
trash");
    }
    if (debug) IJ.write("Reason: V1=" + s.getV1() + ", V2=" + s.getV2() +
"; orientation=" + s.isOrientation());
    }
    if (debug) {
        IJ.write(" ");
        IJ.write(" ");
        IJ.write("finished - end!");
        IJ.write("LSG: start=" + sol_start + ", end=" + sol_end);
    }

    // =====
    // GENERATES IMAGES FOR DOMINO LISTS (sol, trash)
    // =====

    n_lsg = sol.size();
    n_trash = trash.size();

    // solution output image:
    String title_sol = "solution";
    int fill_sol = NewImage.FILL_WHITE;
    result = NewImage.createByteImage(title_sol, n_lsg * (W_IDEAL + offset_x) + 2
* border, H_IDEAL + 2 * border, 1, fill_sol);
    result_ip = result.getProcessor();

    // Inserts suitable DStones in the output image:
    Iterator k = sol.iterator();
    int count = 0; // counts the stones
    while (k.hasNext()) {
        DStone tmp = (DStone)k.next();
        ImageProcessor tmp_ip = tmp.getImg().getProcessor();
        tmp_ip = tmp_ip.resize(H_IDEAL, W_IDEAL);
        if (tmp.isOrientation()) {
            tmp_ip = tmp_ip.rotateLeft();
        } else {
            tmp_ip = tmp_ip.rotateRight(); // orientation == false
        }
        result_ip.insert(tmp_ip, count * (W_IDEAL + offset_x) + border, border);
        count++;
    }

    // trash output image:
    String title_trash = "trash";
    int fill_trash = NewImage.FILL_WHITE;
    resultTrash = NewImage.createByteImage(title_trash, n_trash*(H_IDEAL +
offset_x) + border * 2, W_IDEAL + 2 * border + textheight, 1, fill_trash);

    resultTrash_ip = resultTrash.getProcessor();

    // Inserts non suitable DStones in the output image:
    if (trash.size() > 0) {
        k = trash.iterator();
        count = 0;
        while (k.hasNext()) {
            DStone tmp = (DStone)k.next();
            ImageProcessor tmp_ip = tmp.getImg().getProcessor();
            tmp_ip = tmp_ip.resize(H_IDEAL, W_IDEAL);
            // tmp_ip = tmp_ip.rotateLeft();
            // inserts stone image:
            resultTrash_ip.insert(tmp_ip, count * (H_IDEAL + offset_x) +
border, border);

            // inserts stone value
            resultTrash_ip.setAntialiasedText(true);
            resultTrash_ip.drawString(" v1 = " + tmp.getV1(), count * (H_IDEAL
+ offset_x) + border, W_IDEAL + border + textheight);
            resultTrash_ip.drawString(" v2 = " + tmp.getV2(), count * (H_IDEAL
+ offset_x) + border, W_IDEAL + border + textheight + 10);
            count++;
        }
    }

```

```

    }
  } else {
    IJ.write(" All domino stones match!");
  }

  // Shows output images
  result.show();
  resultTrash.show();
}

}

/**
 * This class represents a point in 2D space and is used for the flood-filling algorithm.
 *
 * @author Andrzej Kozlowski
 *
 * @since 05/2005
 * @version 1.0
 */
class Node
{
  int x,y;

  /**
   * Creates a node with the given x,y-coordinates.
   *
   * @author Andrzej Kozlowski
   *
   * @param x the x-coordinate in 2D space
   * @param y the y-coordinate in 2D space
   */
  Node(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
}

/**
 * This class represents a bounding box (the smallest box possible around an entire
 * region) and
 * is used for cropping each domino stone.
 *
 * @author Andrzej Kozlowski
 *
 * @since 05/2005
 * @version 1.0
 */
class BoundingBox
{
  int x1,x2,y1,y2;

  /**
   * Creates a bounding box with the given x,y-coordinates. The initial area size of
   it is 0.
   *
   * @author Andrzej Kozlowski
   *
   * @param x the minimal x-coordinate in 2D space
   * @param y the minimal y-coordinate in 2D space
   */
  BoundingBox(int max_width, int max_height)
  {
    x1 = max_width;
    y1 = max_height;
    x2 = 0;
    y2 = 0;
  }
}

```

```

    }

    /**
     * @return Returns the x1.
     */
    public int getX1() {
        return x1;
    }

    /**
     * @param x1 The x1 to set.
     */
    public void setX1(int x1) {
        this.x1 = x1;
    }

    /**
     * @return Returns the x2.
     */
    public int getX2() {
        return x2;
    }

    /**
     * @param x2 The x2 to set.
     */
    public void setX2(int x2) {
        this.x2 = x2;
    }

    /**
     * @return Returns the y1.
     */
    public int getY1() {
        return y1;
    }

    /**
     * @param y1 The y1 to set.
     */
    public void setY1(int y1) {
        this.y1 = y1;
    }

    /**
     * @return Returns the y2.
     */
    public int getY2() {
        return y2;
    }

    /**
     * @param y2 The y2 to set.
     */
    public void setY2(int y2) {
        this.y2 = y2;
    }
}

```

9.2 DStone.java

```

import java.awt.Rectangle;

import ij.*;
import ij.gui.Roi;
import ij.process.ImageProcessor;

```

```

import ij.gui.NewImage;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import ij.gui.Roi;
import java.awt.Rectangle;

/**
 * This is a virtual model of a domino stone. It contains the two values of a stone
 * as well as a grayscale picture and a binary picture (for the operations as rotating
 * and counting the points).
 * @author Walter Jenner, Andrzej Kozlowski, Bjoern Schnetzinger
 * @since 05/2005
 * @version 1.0
 */
public class DStone
{
    private int v1;
    private int v2;
    private ImagePlus img;
    private ImagePlus binary;
    private boolean orientation; /* necessary for method sortStones(). if false,
                                stone must be rotated
180 degree to match the domino row! */
    /* debug console output */
    boolean debug = false;

    /**
     * Creates a DStone object with 2 ImagePlus parameters. v1 and v2 are initially
     set to -1, as
     * the true values (from 0 to 6) have yet to be determined. the variable
orientation is important
     * for the final output picture.
     * @param im the DStone grayscale image
     * @param bin the DStone binary image used for operations
     */
    public DStone(ImagePlus im, ImagePlus bin)
    {
        v1 = -1;
        v2 = -1;
        img = im;
        binary = bin;
        orientation = true; // true means no changes, false means stone must be
rotated 180 degrees.
    }

    /**
     * DESCRIPTION:
     *
     * setPoints() finds out the number of white points on both halves of one
dominostone
     * and sets the class-member values: v1, v2.
     *
     * CONFIGURATION:
     *
     * (1) Specify size:
     * Size of the image of the domino stone that is used for white-points-
recognition.
     *
     * (2) The template coordinates:
     * It (an array t) contains all coordinates (u, v) of the centers where white
points
     * (on a dominostone) may occur. (9 positions in total). Must match the specified
size.
     *
     * (3) Repetition of dilate, erode:
     * It depends on the size of an image (of a domino-stone) that its white points
can be found.
     * Therefore this method can eliminate white noise (dilate) and open (erode) the
white points on that image.

```

```

    * Only then it is possible to archive good results finding white points with the
    template.
    * Therefore you can adjust the repetition for the dilate and erode process in
    dilate_rep and erode_rep.
    *
    * WORKING PRINCIPLE:
    *
    * The image that contains one domino stone is resized to a specified resolution.
    Dependent on that
    * size it does the white-points-recognition: checkPoints(). That generates an
    array which contains all
    * flags of the corresponding coordinates of the template (template found: tf).
    findSetValues() identifies this combination of
    * the flags and recognizes the values of both sides of a domino stone (between 0
    and 6).
    *
    *
    * SUBMETHODS that are necessary:
    *
    * checkPoints()
    * showTemplateCoordinates()
    * findSetValues()
    *
    * @author Björn Schnetzinger
    *
    * @param showDebugImages if true: for showing the domino stone images that where
    optimized for this method
    (the 9 centers of the template are shown as crosshairs too)
    *
    */
    public void setPoints(boolean showDebugImages)
    {
        // =====
        // CONFIGURATION
        // =====

        // specified image size (in pixel) of domino stone:
        int W_IDEAL = 53;
        int H_IDEAL = 100;

        // template that contains the coordinates (u, v) of the centers where
        white points may occur.
        int[][] t = new int[][]{
            {14, 13}, // position 1
            {27, 13}, // position 2
            {42, 13}, // ...
            {14, 26},
            {27, 26},
            {42, 26},
            {14, 39},
            {27, 39}, // ...
            {42, 39}}; // position 9

        // repetition of dilate and erode:
        int dilate_rep = 0;
        int erode_rep = 4;
        // 8 dominostones (1):
        //int dilate = 2;
        //int erode = 5;

        // =====
        // DECLARATION OF VARIABLES
        // =====

        // tf ... template found
        boolean[] tf = new boolean[9];
        ImageProcessor ip = (this.binary).getProcessor();

        ImagePlus n_image; // image, that is optimized for white-point-recognition
        ImageProcessor n_ip;

```

```

        ImagePlus result;

        // =====
        // PREPARING IMAGE FOR WHITE-POINT-RECOGNITION
        // =====

        /* Makes a copy of class-member image (ip) for changes, and resizes it to the
        specified size.
           Used for applying dilate/erode: */
        String title = "copy for resizing and counting its points";
        int fill = NewImage.FILL_BLACK;
        n_image = NewImage.createByteImage(title,W_IDEAL,H_IDEAL,1,fill);
        n_ip = n_image.getProcessor();
        n_ip = ip.resize(W_IDEAL, H_IDEAL);

        /* eliminates white noise (dilate) and opens (erode) the white points of a domino
        stone. Only then it is possible to archive
           good results finding white points with the template: */

        for (int i = 1; i <= dilate_rep; i++) {
            n_ip.dilate();
        }
        for (int i = 1; i <= erode_rep; i++) {
            n_ip.erode();
        }

        // =====
        // FIND WHITE POINTS WITH TEMPLATE
        // =====

        if (debug) {
            IJ.write(" ");
            IJ.write("starting to check 1nd half ....");
        }
        // FIRST HALF:
        tf = checkPoints(n_ip, t);
        findSetValues(tf);

        if (showDebugImages) n_ip = showTemplateCoordinates(n_ip, t);

        n_ip.flipVertical(); //Flips the image or ROI vertically.

        // SECOND HALF:
        if (debug) IJ.write("starting to check 2nd half ....");

        tf = checkPoints(n_ip, t);
        findSetValues(tf);

        if (showDebugImages) n_ip = showTemplateCoordinates(n_ip, t);

        n_ip.flipVertical(); // undoes previous flip.

        if (showDebugImages) {
            result = new ImagePlus("FINAL PICTURE",n_ip);
            result.show();
        }
        if (debug) IJ.write(" ");
    }

    /**
     * Submethod of setPoints().
     *
     * With the information where white points can occur (the coordinates of the
    template), this
     * method checks on which positions white points in the image of the stone occur.
     *
     * if a white point is found on the position of the template, then the
    correspondent flag in the boolean array
     * is set true.
     *
     */

```

```

* @author Björn Schnetzinger
*
* @param ip ImageProcessor of a Stone where should be searched for white points
* @param t array that contains u and v coordinates of a template
* @return boolean array for the flags that correspond to the template
*/
private boolean[] checkPoints(ImageProcessor ip, int[][] t) {
    boolean[] tf = new boolean[9];

    for (int i=0; i < t.length; i++) {
        int p = ip.getPixel(t[i][0],t[i][1]); // gets the coordinates of
the possible white points
        if (p >= 200) { // if this pixel is white (or nearly white)

            tf[i] = true;
        } else {
            tf[i] = false;
        }
        if (debug) IJ.write(" No.:" + i + " " + tf[i] + "; check coords:
" + t[i][0] + "/" + t[i][1] + "; check value = " + p + ";");
    }
    return tf;
}

/**
* shows the crosshairs for all the coordinates of the template in a certain color
*
* @author Björn Schnetzinger
*
* @param ip ImageProcessor of a DStone
* @param t template
* @return altered image (with crosshairs)
*/
private ImageProcessor showTemplateCoordinates(ImageProcessor ip, int[][] t) {
    int color = 125;
    for (int i=0; i < t.length; i++) {
        ip.putPixel(t[i][0]-2,t[i][1], color);
        ip.putPixel(t[i][0]-1,t[i][1], color);
        ip.putPixel(t[i][0]+1,t[i][1], color);
        ip.putPixel(t[i][0]+2,t[i][1], color);

        ip.putPixel(t[i][0],t[i][1], color);

        ip.putPixel(t[i][0],t[i][1]+3, color);
        ip.putPixel(t[i][0],t[i][1]+2, color);
        ip.putPixel(t[i][0],t[i][1]+1, color);
        ip.putPixel(t[i][0],t[i][1]-1, color);
        ip.putPixel(t[i][0],t[i][1]-2, color);
        ip.putPixel(t[i][0],t[i][1]-3, color);
    }
    return ip;
}

/**
* identifies the flags that has been set for the template coordinates:
*
* with the already found white points, it searches for patterns that specify the
* right values of one side of a domino stone
*
* @author Björn Schnetzinger
*
* @param tf that contains the boolean flags which correspond to the coordinates
of the template
*/
private void findSetValues(boolean[] tf){

    int value;
    // tf[0] means that field is white
    // !tf[0] means that field is black
    if (!tf[0] && !tf[2] && !tf[4] && !tf[6] && !tf[8]) {
        value = 0;
    }
}

```

```

        setValue(value);
    if (debug) IJ.write("Wert gefunden: " + value);
} else if (!tf[0] && !tf[2] && tf[4] ) {
    value = 1;
    setValue(value);
    if (debug) IJ.write("Wert gefunden: " + value);
} else if (tf[2] && tf[6] && !tf[0] && !tf[4] ||
           tf[0] && tf[8] && !tf[2] && !tf[4]) {
    value = 2;
    setValue(value);
    if (debug) IJ.write("Wert gefunden: " + value);
} else if (tf[0] && tf[4] && tf[8] && !tf[2] ||
           tf[2] && tf[4] && tf[6] && !tf[0]) {
    value = 3;
    setValue(value);
    if (debug) IJ.write("Wert gefunden: " + value);
} else if (tf[0] && tf[2] && tf[6] && tf[8] && !tf[3] && !tf[4]) {
    value = 4;
    setValue(value);
    if (debug) IJ.write("Wert gefunden: " + value);
} else if (tf[0] && tf[2] && tf[4] && tf[6] && tf[8]) {
    value = 5;
    setValue(value);
    if (debug) IJ.write("Wert gefunden: " + value);
} else if (tf[0] && tf[2] && tf[3] && tf[5] && tf[6]) {
    value = 6;
    setValue(value);
    if (debug) IJ.write("Wert gefunden: " + value);
} else {
    if (debug) IJ.write("Wert NICHT gefunden! ");
    value = -1;
    setValue(value);
}
}

/**
 * sets the value for one side of a stone. if the value for the first side has
been
 * already set, it sets the 2. value.
 *
 * @author Björn Schnetzinger
 *
 * @param v
 */
private void setValue(int v) {
and set
    if (this.v1 == -1) { // if first value of stone has not been already found

        this.v1 = v;
        if (debug) IJ.write("v1 = " + this.v1);
    } else {
        this.v2 = v;
        if (debug) IJ.write("v2 = " + this.v2);
    }
}

/**
 * @return Returns the orientation.
 */
public boolean isOrientation() {
    return orientation;
}

/**
 * @param orientation The orientation to set.
 */
public void setOrientation(boolean orientation) {
    this.orientation = orientation;
}

```

```

/**
 * @author walter jenner, www.teesacker1.at
 * @version 1.0
 *
 * Rotates an already cropped Domino Stone Image, that the longer edge is
vertical.
 * The resulting Image is cropped again. Both Images, the original and the binary,
are rotated
 */
public void rotate()
{
    ImageProcessor ipBin= this.binary.getProcessor();
    ImageProcessor ipBinDup= ipBin.duplicate();

    ImageProcessor ipOrig= this.img.getProcessor();
    ImageProcessor ipOrigDup= ipOrig.duplicate();

    int width= ipBinDup.getWidth();
    int height= ipBinDup.getHeight();

    if (width > height){
        //rotate binary image 90 degrees
        ImageProcessor ipTemp= ipBinDup.rotateLeft();
        this.binary = new ImagePlus("Binary Stone Image",ipTemp);
        ipBinDup= this.binary.getProcessor();

        //rotate original image too
        ImageProcessor ipTemp2= ipOrigDup.rotateLeft();
        this.img = new ImagePlus("Stone Image",ipTemp2);
        ipOrigDup= this.img.getProcessor();

        //reset width and height
        width= ipBinDup.getWidth();
        height= ipBinDup.getHeight();

    }

    int x= 0;
    int y= 0;

    double a= 0;
    double b= 0;

    double alpha;

    //Find the rotating angle
    for (int i= 0; i<width; i++){
        if (ipBinDup.getPixel(i,0)==0){
            x= i;
            break;
        }
    }
    for (int j= 0; j<height; j++){
        if (ipBinDup.getPixel(0,j)==0){
            y= j;
            break;
        }
    }

    if (y>4&&x>4){
        alpha= java.lang.Math.toDegrees(java.lang.Math.atan2(y,x));
        a= Math.sqrt(Math.pow(x,2)+Math.pow(y,2));
        b= Math.sqrt(Math.pow(width-x,2)+Math.pow(height-y,2));

        if (a>b){
            alpha= alpha+ 90;
        }

        ipBinDup.rotate(alpha);
        this.binary= new ImagePlus("Binary Image",ipBinDup);
    }
}

```

```

        Roi roi= getCropRoi();

        ipBinDup.setRoi(roi);
        ImageProcessor ip2 = ipBinDup.crop();
        this.binary = new ImagePlus("Binary Stone Image",ip2);

        ipOrigDup.rotate(alpha);
        ipOrigDup.setRoi(roi);
        ImageProcessor ip3 = ipOrigDup.crop();

        this.img = new ImagePlus("Stone Image",ip3);
    }
}

/**
 * @author walter jenner, www.teesacker1.at
 * @version 1.0
 *
 * Finds a ROI which is used for cropping the rotated Domino Stone.
 * Runs from each edge towards the center, and when the first black pixel appears,
 * the crop edge is set.
 *
 * @return The ROI that can be used for cropping the Domino Stone
 */
public Roi getCropRoi(){
    Roi roi;

    ImageProcessor ipBin= this.binary.getProcessor();

    int height= ipBin.getHeight();
    int width= ipBin.getWidth();
    int xIn= ipBin.getWidth(), xOut= 0, yIn= height, yOut= 0;

    for (int y= 0; y< height; y++){
        for (int x= 0; x< width; x++){
            if (ipBin.getPixel(x,y) == 0){
                if (x < xIn){
                    xIn = x;
                }
                if (x > xOut){
                    xOut = x;
                }
                if (y < yIn){
                    yIn = y;
                }
                if (y > yOut){
                    yOut = y;
                }
            }
        }
    }

    roi= new Roi(xIn, yIn, xOut-xIn, yOut-yIn);

    return roi;
}

/**
 * @return Returns the img.
 */
public ImagePlus getImg() {
    return img;
}

/**
 * @param img The img to set.
 */
public void setImg(ImagePlus img) {
    this.img = img;
}

/**

```

```
    * @return Returns the v1.
    */
    public int getV1() {
        return v1;
    }
    /**
     * @param v1 The v1 to set.
     */
    public void setV1(int v1) {
        this.v1 = v1;
    }
    /**
     * @return Returns the v2.
     */
    public int getV2() {
        return v2;
    }
    /**
     * @param v2 The v2 to set.
     */
    public void setV2(int v2) {
        this.v2 = v2;
    }
}
```