

Theodor-Heuss-Gymnasium Göttingen

Lehrer: Herr Aschenbrenner

Informatik

Steganographie mit Bildern

Ascii-Zeichenfolgen in PNG-Dateien verstecken

Projektarbeit

von David Söding

12. März 2023

Inhaltsverzeichnis

1 Ideenfindung	2
2 Zielsetzung	2
3 Vorgehen	2
3.1 Auswertung des ersten Tests	4
3.2 Weitere Tests	5
3.3 Entscheidung über die Anzahl der Informations-Bits	7
4 Implementierung	8
4.1 Probleme	9
4.2 Auswertung	9

1 Ideenfindung

Die erste Idee für dieses Projekt war es, ein Bild-Komprimierungsverfahren in Java zu implementieren, welches PNG-Dateien einliest, aus vier benachbarten Pixeln den mittleren Farbwert bildet, und so ein Bild mit der Hälfte der Dimensionen, also einem Viertel der Pixel, ausgibt. Dies klappte gut, jedoch war das Programm schon nach einer Doppelstunde fertig. Aufgrund dessen entschloss ich mich dazu, eine andere Idee zu verwirklichen. Trotzdem ist der kommentierte Code der ursprünglichen Idee beigelegt, es wird aber nicht weiter darauf eingegangen.

Ich schlug einem Mitschüler, welcher noch in der Ideenfindung war, vor, per Manipulation der Farbwerte von Pixeln Informationen in einer Bilddatei zu verstecken. Als ich jedoch bemerkte, dass ich den Umfang meiner eigenen Idee überschätzt hatte, machte ich diese zweite Idee zu meinem neuen Projekt.

2 Zielsetzung

Ziel der neuen Idee ist es, ASCII-Zeichenketten, welche pro Zeichen acht Bits beanspruchen, in einer PNG-Datei zu verstecken. Verwendet werden soll ein beliebiges Foto. Es sollen die Farbwerte der Pixel so verändert werden, dass Informationen darin gespeichert werden können, das ursprüngliche Bild aber nicht auffällig verändert wird. Diese Idee basiert auf dem Konzept der Steganographie, bei dem sich mit der verborgenen Speicherung oder Übermittlung von Informationen in einem Trägermedium beschäftigt wird.

3 Vorgehen

PNGs mit einem RGB-Farbprofil besitzen vier Farbkanäle. Jeweils einen Farbkanal für rot, grün und blau, sowie einen Kanal für die Transparenz (Alpha). Jeder Pixel hat gewöhnlich eine Farbtiefe von acht Bits pro Farbkanal, also 32 Bits pro Pixel. Beansprucht man nun pro Farbkanal die letzten paar Bits für die Informationsübertragung, so bleibt die Farbe ungefähr gleich.

Dementsprechend muss zuallererst herausgefunden werden, wie viele der acht Bits genutzt werden können, ohne dass das Bild sichtbar an Qualität verliert. Dafür werden die letzten n Bits pro Farbkanal mit Zufallszahlen ersetzt und das ausgegebene Bild anschließend begutachtet.

Hierfür wurde folgender Code geschrieben:

```

import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import java.io.File;
import java.util.concurrent.ThreadLocalRandom;

public class HiddenMessage{
    public static void main(String args[]) {
        //Aufrufen der Methode mit Eingabe- und Ausgabepfad
        hideMessage("parrot.png", "parrotTest0.png");
    }

    public static void hideMessage(String inputPath, String outputPath){
        try {
            //Importieren des Bildes
            BufferedImage inputImg = ImageIO.read(new File(inputPath));
            //Fuer jeden Pixel werden fuer jeden Kanal die letzten 3 bits erst auf
            //null gesetzt und dann mit einem 32bit Zufalls-Integer kombiniert
            for (int x = 0; x < inputImg.getWidth(); x++) {
                for (int y = 0; y < inputImg.getHeight(); y++) {
                    int color = inputImg.getRGB(x, y);
                    int randomNum = ThreadLocalRandom.current().nextInt();
                    color = (color & 0xF8F8F8F8) | (randomNum & ~0xF8F8F8F8);
                    inputImg.setRGB(x, y, color);
                }
            }
            //Speichern des veraenderten Bildes im Ausgabepfad
            ImageIO.write(inputImg, "png", new File(outputPath));
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

Importiert wird `ImageIO` und `File`, um Bilddateien lesen und schreiben zu können. `BufferedImage` dient dem Auslesen und Verändern der Farbwerte von Pixeln des Bildes. `ThreadLocalRandom` wird genutzt um, eine Zufallszahl zu erzeugen.

Die beiden verschachtelten `for`-Schleifen iterieren über jedes Pixel. Für jedes Pixel wird der Farbwert ausgelesen und ein zufälliger `int` erzeugt. Da `ints` stets 32 Bits nutzen, wurde also eine zufällige Folge an 32 Bits generiert. In diesem Code-Beispiel werden mit `(color & 0xF8F8F8F8)` die letzten drei Bits pro Kanal auf null gesetzt. `0x` sagt Java, dass die folgende Zahl im Hexadezimalsystem angegeben ist. `F8F8F8F8` in hexadezimal ist in binär `1111000111100011110001111000`. Diese 32 Bits ergeben sich durch vier mal fünf Einsen, gefolgt von drei nullen. Durch eine `&`-Operation mit dem Farbwert des Pixels wird jedes Bit des Farbwertes mit dem entsprechenden Bit der Hexadezimal-Zahl einzeln per `&`-Operation zu einem neuen Bit kombiniert. Die Bits des Farbwertes, die mit einer Eins kombiniert werden, bleiben unverändert, während die Bits, die mit null kombiniert werden, auf null gesetzt werden. Es wurden

also für jeden Achter-Block die letzten drei Bits auf null gesetzt.

Mit (`randomNum & ~0xF8F8F8F8`) werden bei der Zufallszahl die ersten fünf Bits pro Kanal auf null gesetzt, da `~` ein logisches nicht ist. Mit `|` (oder) werden die beiden Bitfolgen kombiniert.

Dies wird für jedes Pixel durchgeführt, wonach das Bild als PNG-Datei gespeichert wird.

3.1 Auswertung des ersten Tests

Für das Experiment wurde ein kostenloses Stockfoto eines Papageien verwendet (siehe Abb. 1 links). Bei dieser Eingabe liefert der oben beschriebene Code das rechte Bild aus Abb. 1 als Ausgabe.



Abb. 1: Links: Foto eines Papageien; rechts: Ausgabe des Codes.

Im rechten Bild lässt sich ein eindeutiger Qualitätsunterschied erkennen. Das Auftreten von Color banding (siehe Abb. 2) war für mich verwunderlich.

Mithilfe von Photoshop lassen sich die Unterschiede der beiden Bilder veranschaulichen. Wenn beide Bilder übereinander gelegt werden und der Mischmodus der oberen Ebene auf Subtrahieren gesetzt wird, ergeben sich neue Farbwerte, die die Differenz der Pixel darstellen. Dieses Bild ist schwarz, da beide Bilder nur leicht voneinander abweichen. Mithilfe der Einstellungsebene **Curves** können die Farbwerte von 0 bis 4 auf 0 bis 255 verteilt werden (siehe Abb. 3), wodurch die Differenzen sichtbar werden (siehe Abb. 4).

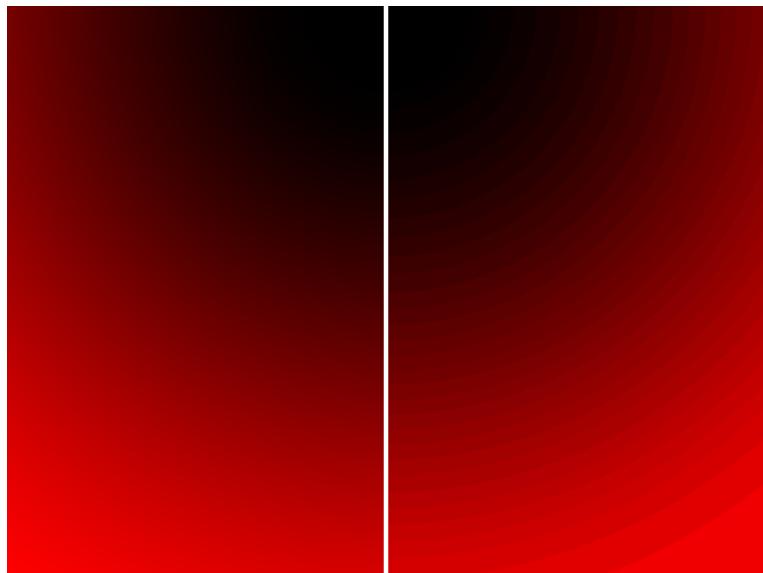


Abb. 2: Beispiel von Color banding mit einem Gradienten einer Farb-Tiefe von 8 Bits (links) und 5 Bits (rechts). ©David Söding, 2022

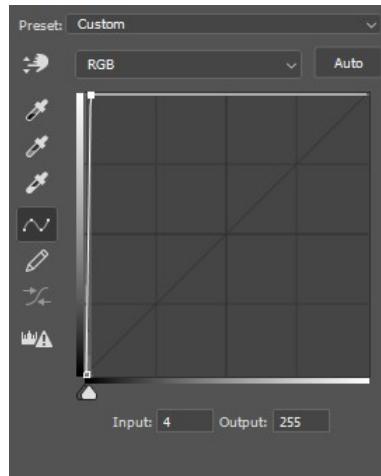


Abb. 3: Einstellungsebene Curves.

Auch hier sind neben dem Rauschen auch Color banding Artefakte zu erkennen. Ich erwartete nur Rauschen, da die drei Bits zufällig gesetzt werden, also keine Muster entstehen sollten. Warum diese aber trotzdem auftreten lässt sich einfach erklären, wenn man im Code die `oder`-Operation weglässt, also die letzten drei Bits auf null lässt. Wie sich in Abb. 5 erkennen lässt, ist das die Color banding nun noch besser sichtbar. Dass sie hier entstehen ist logisch, da die letzten drei Bits abgerundet werden. Diese Farbwerte werden dann per `oder` mit den die Zufalls-Bits kombiniert, wodurch das Bild mit F8F8F8F8 mehr Rauschen hat, das Color banding aber trotzdem noch sichtbar ist.

3.2 Weitere Tests

Die Veränderung, die durch die Manipulation der drei letzten Bits verursacht wird, ist nicht mehr unauffällig. Deshalb werden mit einer Änderung der Hexadezimalzahl zu

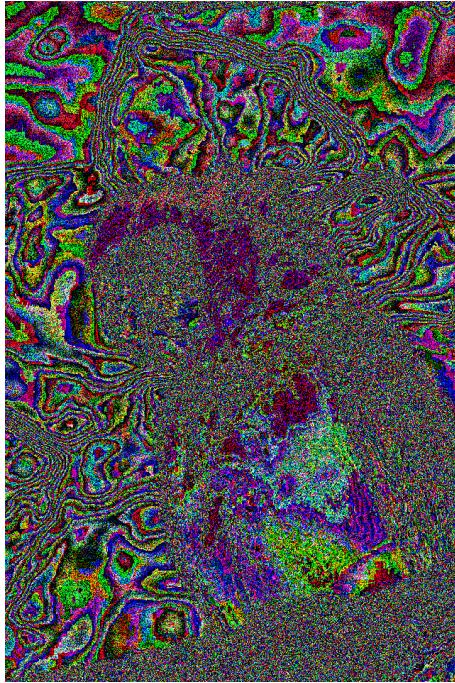


Abb. 4: Differenz der Farbwerte in Photoshop sichtbar gemacht.

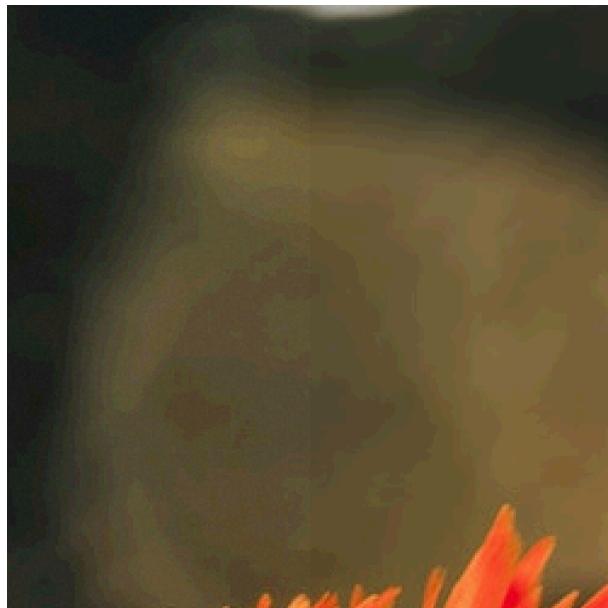


Abb. 5: Links: Zufallswert für die letzten 3 Bits; rechts: letzten 3 Bits = 0.

FCFCFCFC für zwei Bits sowie FEEFEFEF für ein Bit zwei weitere Bilder betrachtet (siehe Abb. 6).

Wie sich in Abb. 7 erkennen lässt, sind bei zwei Informations-Bits noch sichtbare Änderungen vorhanden. Sowohl Rauschen als auch Color banding sind erkennbar. Mit einem Informations-Bit hingegen sind das Original und die Ausgabe nahezu identisch. Mit Photoshop wurde auch für ein Informations-Bit mit dem gleichen Prozess ein Differenz-Bild erzeugt (siehe Abb. 9).



Abb. 6: Links: 2 Informations-Bits; rechts: 1 Informations-Bit.



Abb. 7: Links: 2 Informations-Bits; Mitte: 1 Informations-Bit; rechts: Original.

3.3 Entscheidung über die Anzahl der Informations-Bits

Wählen wir ein Informations-Bit pro Farbkanal, so ergeben sich vier Bits pro Pixel. Da die ASCII Zeichencodierung acht Bits nutzt, kann also ein ASCII-Zeichen in zwei Pixeln gespeichert werden. Das Bild des Papageien hat eine Auflösung von 640×960 . Dies entspricht ca. 600 000 Pixeln und somit 300 000 Zeichen. Dies sollte für die meisten Anwendungszwecke reichen. Es könnten sogar kurze Romane codiert



Abb. 8: Differenz der Farbwerte mit einem Informations-Bit in Photoshop sichtbar gemacht.

werden, obwohl das Bild aus heutiger Sicht eine vergleichsweise geringe Auflösung hat.

Das gesamte englischsprachige Skript von Shrek hat eine Länge von 93 952 Zeichen, das Skript von Shrek 2 hat eine Länge von 94 315 Zeichen und das Skript von Shrek the Third hat 122 882 Zeichen. Addiert ergibt das 311 149 Zeichen. Wird die Auflösung des Bildes von 640×960 auf 649×973 erhöht, so können 315 738 Zeichen gespeichert werden. Ziel ist es nun, alle drei Skripts in das Bild des Papageien zu codieren.

4 Implementierung

In dem Programm wird eine ähnliche Herangehensweise genutzt, wie im vorherigen Code. Die Methode hat nun drei Parameter: das Eingabe-Bild, die zu codierende Nachricht und den Pfad für das Ausgabe-Bild. Statt zwei `for`-Schleifen für die Koordinaten des Pixels werden jetzt aber mit einer einzigen Variable gezählt und aus dieser Variable durch die selbst geschriebene Methode `numToCoords` die Koordinaten berechnet. Außerdem werden die Bits der Farbwerte nicht mit Zufallszahlen, sondern mit den ASCII-Werten der Zeichen ersetzt. Zusätzlich wurde ein \an das Ende der Nachricht angehängt, um beim Auslesen der Nachricht zu wissen, wann diese zu Ende ist. Das Auslesen funktioniert ähnlich und bedarf daher keiner weiteren Erklärung.

4.1 Probleme

Es traten zwei nennenswerte Probleme bei der Implementierung auf:

- Zu Beginn wurde die Nachricht mithilfe eines Scanners eingelesen. Dieser hörte allerdings mitten im Textdokument auf zu lesen. Dies scheint ein Bug zu sein, weswegen stattdessen ein BufferedReader verwendet wurde.
- Der standard Farb-Typ eines über `ImageIO.read` importieren `BufferedImage` ist RGB statt ARGB. Deswegen wurden anfangs die Änderungen des Alpha-Kanals nicht im Bild gespeichert. Es ist nicht möglich, den Farb-Typen eines `BufferedImage` zu verändern. Aufgrund dessen wurde ein weiteres `BufferedImage` mit dem Typ ARGB erstellt, in dieses dann der Inhalt des importierten Bildes per `Graphics2D` kopiert wurde.

4.2 Auswertung

Das exportierte Bild lässt sich mit bloßem Auge nicht vom Original unterscheiden. Lediglich mit Photoshop kann wieder die Differenz sichtbar gemacht werden (siehe Abb. 9). Dadurch ist außerdem zu erkennen, ab welchem Pixel die Nachricht aufhört. Die letzten 15 Zeilen sind schwarz, weil die Nachricht kürzer war als die Anzahl der Pixel geteilt durch zwei. Auch das Auslesen der Nachricht aus dem Bild funktioniert wie gewollt.

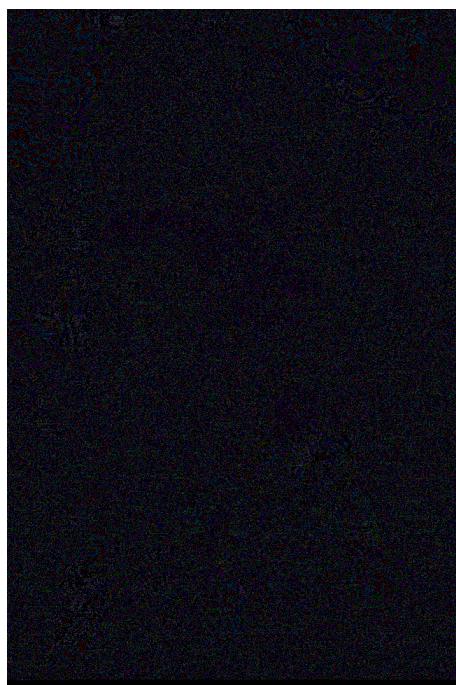


Abb. 9: Differenz der Farbwerte des Bildes mit codierter Nachricht.