

LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME  
**Aspekte der systemnahen Programmierung  
bei der Spieleentwicklung**

Gruppe 127 – Abgabe zu Aufgabe A200  
Wintersemester 2019/20

Leon Brooks

Qiyuan Zhu

Jiesheng Ding

## Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Lösungsansatz</b>	<b>2</b>
2.1 Rahmenprogramm . . . . .	2
2.2 Assembler . . . . .	3
<b>3 Korrektheit</b>	<b>5</b>
<b>4 Performanzanalyse</b>	<b>6</b>
<b>5 Zusammenfassung und Ausblick</b>	<b>9</b>

## 1 Einleitung

Im Rahmen des Praktikums: Aspekte der systemnahen Programmierung bei der Spieleentwicklung war das Ziel dieses Projekts die Implementierung und Optimierung eines Laplace-Filters in Assembler. Als technische Grundlage diente das Raspberry Pi[7] Cluster der Technischen Universität München, welches die für die Implementierung notwendigen Prozessoren in ARMv8-A Architektur[4] zur Verfügung stellte.

Der Laplace-Filter ist ein Filter, der zur Kantendetektion in Bildern verwendet wird. Er basiert auf der Faltung einer Matrix über das Eingabebild. Dafür wird meist folgende Faltungsmatrix angewandt[3]:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad (1)$$

Was dies anschaulich bedeutet ist, dass für jeden Pixel die vier direkt benachbarten Pixel betrachtet und gegen den mittleren gewichtet werden. Dabei erhält der Mittlere die Gewichtung -4, während die umliegenden einfach gewichtet werden. Somit kann die

Faltung auf die Formel

$$A_{(x,y)}^F = \sum_{i=-k}^k \sum_{j=-k}^k M_{(k+i,k+j)} \cdot E_{(x+i,y+i)}^F \quad (2)$$

heruntergebrochen werden[3].  $A_{(x,y)}^F$  bezeichnet einen Farbkanal des Ausgabepixels  $(x,y)$ ,  $M_{(k+i,k+j)}$  den Wert der Faltungsmatrix in der i-ten Spalte der j-ten Reihe und  $E_{(x+i,y+i)}^F$  den gleichen Farbkanal des Pixels  $(x+i, y+i)$  im Eingabebild.

Zusätzlich zur Implementierung des Filters selbst in Assembler war es von Nöten ein entsprechendes Rahmenprogramm in C zu entwerfen, welches in der Lage war mit dem BMP-Format für Bilddateien umzugehen. Insbesondere spielte dabei das korrekte Auslesen der verschiedenen BMP-Header[6] und Bilddaten eine wichtige Rolle, um die Bilder ordnungsgemäß öffnen und wieder abspeichern zu können.

Letztlich sollte auch die Optimierung der Assembler Implementierung und ein entsprechender Benchmark im C Rahmenprogramm Teil des Projekts sein. Hierfür wurde in der Aufgabenstellung auch explizit auf die Möglichkeit der SIMD-Vektoroperationen der ARMv8-A Architektur[2, Kapitel C7] hingewiesen.

## 2 Lösungsansatz

### 2.1 Rahmenprogramm

Das Rahmenprogramm ist von recht kleinen Umfang und kann grob in drei Teile aufgeteilt werden: Das interpretieren der Kommandozeilenargumente, das Laden und Speichern der BMP-Dateien und das ausführen des Assemblercodes und eins optionalen Benchmarks.

Für die Wahl der Übergabeparameter, wurde Wert auf die einfache Bedienung des Programms ohne die Aufgabe der Funktionalität gelegt. Dabei nimmt die Main-Methode folgende Argumente entgegen:

1. Den Pfad zum Eingabebild im BMP-Format oder -h/-help für eine Ausgabe des Manuals zu diesem Programm
2. -b, einem optionalen Flag, der die Verwendung im Benchmark-Modus signalisiert
3. bei Verwendung im Benchmark-Modus, die Anzahl an Wiederholungen, die zwischen den beiden Zeitmesspunkten durchgeführt werden sollen

Folglich sähen korrekte Aufrufe des Programms folgendermaßen aus:

```
1 ./laplace path {-b iterations}
```

oder

```
1 ./laplace -h bzw ./laplace --help
```

Im ersten Teil des Rahmenprogramms wird geprüft ob es sich um einen -h/-help Aufruf handelt bzw. ob der angegebene Dateipfad korrekt ist und eventuell entsprechende Fehlermeldungen ausgegeben. Zusätzlich wird geprüft, ob das -b Flag gesetzt wird und es sich bei der Anzahl von Wiederholungen um einen gültigen int Wert handelt.

Sind alle eingaben gültig, werden die BMP-Header mit Hilfe entsprechender Structs ausgelesen und zwischengespeichert. Anschließend wird genügend Speicher für die doppelte Bytegröße des Bildes alloziert. Dies ermöglicht es dem Assemblerprogramm die Ergebnisse in die untere Hälfte des allozierten Speichers zu schreiben. Dadurch können einfach beliebig viele Wiederholungen im Benchmark Modus durchgeführt werden, ohne die Eingabedaten zu überschreiben.

Handelt es sich um einen Aufruf ohne Benchmark, wird nun die SIMD-optimierte Version der Assemblerfunktion des Laplace-Filters aufgerufen und anschließend die Ergebnisse in einer neuen BMP-Datei mit den gleichen Headern der Eingabedatei abgespeichert. Ist jedoch der Benchmark Modus aktiviert, so wird mithilfe der CLOCK\_MONOTONIC zunächst ein Zeitmesspunkt erstellt, dann die nicht optimierte Variante des Filters so oft ausgeführt, wie vom Benutzer angegeben, wieder ein Zeitmesspunkt erstellt und schließlich die Differenz berechnet und abgespeichert. Das Gleiche Verfahren wird nun für die optimierte Variante durchgeführt und am Ende die Ergebnisse auf der Konsole ausgegeben. Letztlich wird auch in diesem Modus ein Ausgabebild erzeugt. Hierbei wurde insbesondere auf die Verwendung von CLOCK\_MONOTONIC geachtet und dass keine anderen Aufrufe als die Filterfunktionen selbst zwischen den Messpunkten liegen, um ein möglichst akkurate Ergebnis zu erzielen.

## 2.2 Assembler

Die Implementierung des Laplace-Filters selbst lässt sich in eine standard- und eine SIMD-optimierte Version aufteilen. Beide folgen in ihrer Signatur, bis auf den Rückgabetyp der Angabe:

```
1 extern void laplace(unsigned char* bilddaten, int höhe, int breite)
```

Die Bilddaten haben formell den Typ unsigned char, da jede Farbe im BMP-Format die Größe ein Byte hat und einen Wertebereich von 0-255. Wie schon zuvor erwähnt ist der Rückgabetyp void, da die Funktion die Ergebnisse direkt in die zweite Hälfte des zuvor allozierten Speichers schreibt, um die mehrfach Verwendung hintereinander durch den Benchmarkmodus zu vereinfachen.

Die erste Optimierung, die in beiden Funktionen getroffen wurde, ergibt sich aus der oben genannten Formel<sup>(2)</sup> im Vergleich zur Faltungsmatrix<sup>(1)</sup>. So wird in der Formel jedes Element der Matrix mit dem entsprechenden Pixel, bezüglich dem Mittleren, im Ursprungsbild multipliziert und die Ergebnisse aufaddiert. Da die Faltungsmatrix aber vier mal eine 0 enthält, wären das vier nicht benötigte Leseoperationen und acht nicht benötigte Rechnungen. Aus diesem Grund wurden in beiden Implementierungen nur die direkt benachbarten Pixel betrachtet. Die Standardimplementierung folgt somit diesem vereinfachten Pseudocode:

```

1 // Berechnung wichtiger Größen/Konstanten
2
3 for(i=0; i<höhe; i++){
4     for(j=0; j<(breite*3) /*drei Farben pro Pixel*/; j++) {
5         Farbe_mitte = lade Farbe an der Position [i][j];
6         Farbe_oen, unten, rechts, links;
7
8         if(i == 0 /*oberst Reihe*/){
9             oen = mitte;
10        } else {
11            oen = lade Farbe an Position [i-1][j];
12        }
13
14        if(i == höhe - 1 /*letzte Reihe*/){
15            unten = mitte;
16        } else {
17            unten = lade Farbe an Position [i+1][j];
18        }
19
20        if(j >= (breite*3) - 3 /*letzte Pixel der Reihe*/){
21            rechts = mitte;
22        } else {
23            rechts = lade Farbe an Position [i][j+3];
24        }
25
26        if(j <= 2 /*erste Pixel der Reihe*/){
27            links = mitte;
28        } else {
29            links = lade Farbe an Position [i][j-3];
30        }
31
32        mitte = mitte * (-4) + oen + unten + rechts + links;
33
34        if(mitte < 0) mitte = 0;
35        if(mitte > 255) mitte = 255;
36    }
}

```

```

37     speichere mitte an der Position[i][j];
38 }
39
40 // Jede Spalte wird im BMP_Format am Ende so weit mit nullen aufgefüllt,
41 // bis die Bytegröße der Spalte durch 4 teilbar ist
42 speichere korrekte Anzahl an Padding zeroes;
43 }
```

Die Entscheidung Randwerte mit dem Wert der ursprünglichen Farbe zu ersetzen schien angesichts dessen, das der Algorithmus auf der Gewichtung der umliegenden Pixel basiert am sinnvollsten.

In der SIMD-Optimierung hingegen werden Vektorregister[1, Kapitel 4.6.3] verwendet. Diese besitzen eine maximale Größe von 16 Byte und können in verschiedenen Modi bedient werden. Dabei können sie mit 16 ein Byte großen, bis zu 2 acht Byte großen Elementen befüllt werden (Die Größe hält dabei aber die übliche Form von Zweierpotenzen ein). Obwohl die Eingabedaten pro Farbkanal nur eine Größe von einem Byte haben, können jedoch nur acht gleichzeitig in einem Vektorregister bearbeitet werden. Dies ist darauf zurückzuführen, dass die Werte innerhalb von Rechnungen den 1 Byte Bereich (also 0-255) auch übersteige bzw. darunter liegen können (z.B bei  $\cdot(-4)$ ). Folglich müssen die Werte beim laden mittels uxtl[2, Kapitel C7.2.377] gestreckt und beim Speichern mittels xtn[2, Kapitel C7.2.381] gestaucht werden. Das Clamping auf auf 0 bzw. 255 findet mittels cmge[2, Kapitel C7.2.23] und bif[2, Kapitel C7.2.15] bzw. cmgt[2, Kapitel C7.2.24] und bit[2, Kapitel C7.2.16] statt. Letztlich müssen noch die am Ende jeder Spalte verbleibenden Elemente (Anzahl der Farbkanäle pro Spalte mod 8) einzeln abgearbeitet werden. Eine kleine Optimierung die im Vergleich zur Standardversion noch getroffen wurde, ist die Verwendung von einem anstatt vier Registern für die Elemente oben, unten, rechts und links. Ansonsten arbeitet die SIMD-Variante nach demselben Schema.

### 3 Korrektheit

Die 100%ige Korrektheit eines Bildbearbeitungsalgorithmus ist meist etwas schwierig zu bestimmen, da Fehler in einzelnen Pixeln im Gesamtbild nicht unbedingt auffallen. Insbesondere ist es zusätzlich schwierig im Internet Implementierungen zu finden, die weder externen Bibliotheken benutzen, den Laplace-Filter mit anderen kombinieren, noch andere Faltungsmatrizen verwenden. Auch ließen sich keine vergleichbaren C Implementierungen finden. Schließlich wurde entschieden, ein Implementierung des Laplace-Filters im beliebten Bildbearbeitungsprogramm GIMP[8] zu verwenden, da diese sehr einfach zu handhaben ist. Hierbei ließen sich einige Unterschiede erkennen:



Abbildung 1: von Links nach Rechts: Lena Testbild der Angabe, Ausgabe der Standardimplementierung, Ausgabe der SIMD-Implementierung, Laplace-Filter im Programm GIMP

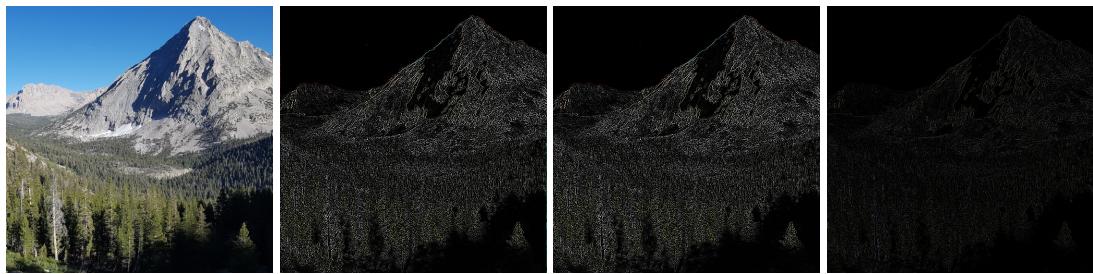


Abbildung 2: wie in 3: Angabe, Standardausgabe, SIMD-Ausgabe, GIMP

So scheint unsere Implementierung zwar allgemein richtig zu sein, jedoch lassen sich über das Ausgabebild verteilte Unsauberkeiten (besonders im Lena Testbild) und etwas grellere Farben erkennen, als das GIMP-Bild aufweist. Dies könnte sich durch eine genauere Faltungsmatrix oder eine vorherige Anwendung eines anderen Filters erklären lassen. Jedoch fanden sich auch nach wiederholter Überprüfung unseres Codes keine Fehler bezüglich der in der Angabe gegebenen Formel<sup>(2)</sup> die diese Unterschiede erklären könnten.

Weitere Testbilder: Abbildung 3-6(leider aufgrund der kleinen Größe hier nicht optimal zu erkennen):

## 4 Performanzanalyse

Da dieses Projekts die Optimierung von Algorithmen zum Ziel hat, ist die Analyse der Laufzeiten der wohl wichtigste Teil dieser Ausführung. Zunächst lässt sich dabei feststellen, dass man von der oben ausgeführten Implementierung<sup>(2.2)</sup> der Assembler Funktionen eine Laufzeit von  $\mathcal{O}(n^2)$  für ein Bild der Größe  $n \times n$  erwarten würde. Dies

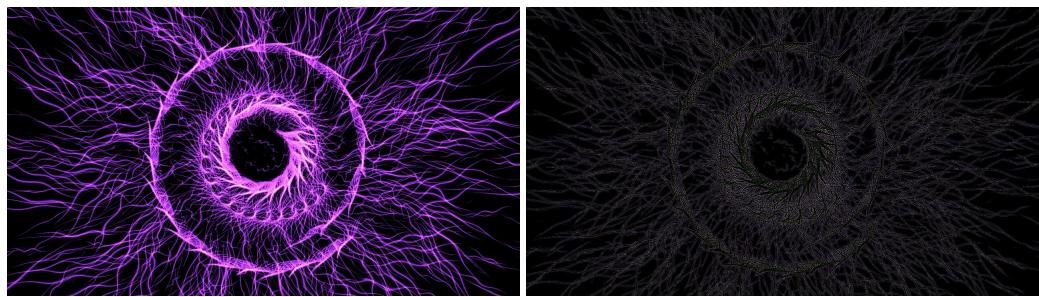


Abbildung 3:  $1280 \times 780$  Bild einer animierten Spiral eines Videos des Youtube-Kanals AA VFX[12]



Abbildung 4:  $3840 \times 2160$  Bild Interstellar des Künstlers Laszlo Magyar(LaczaDesign)[5]

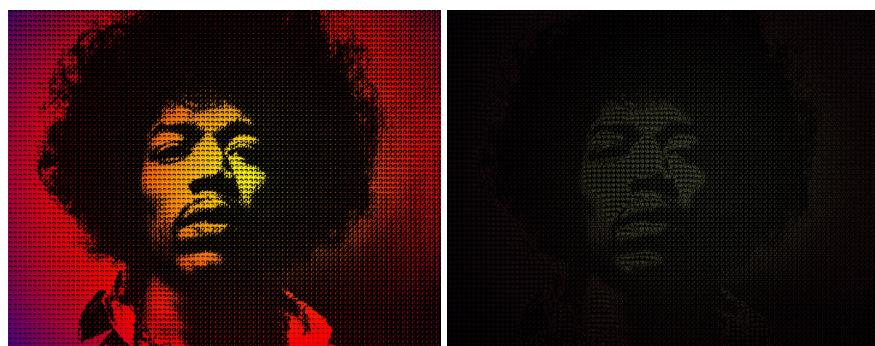
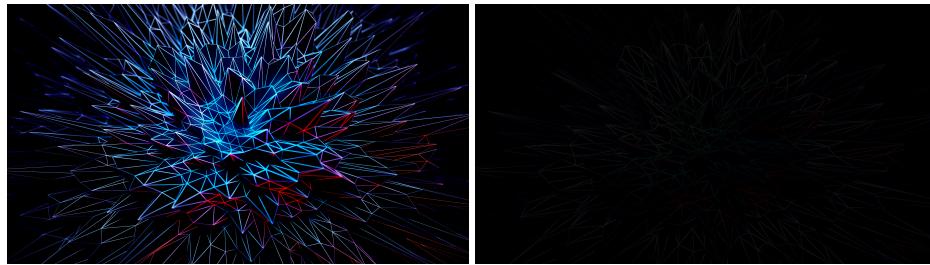


Abbildung 5:  $5400 \times 4236$  Collage des Künstlers Jimmi Hendrix[10]

lässt sich erklären, da für jeden Pixel, von denen es  $n^2$  im Bild gibt, Operationen der Laufzeit  $\mathcal{O}(1)$  (das Betrachten der umliegenden Pixel und die Berechnung des Resultats) ausgeführt werden. Leider sind im Allgemeinen Bilder der Größe  $n \times n$  nicht üblich, weshalb im weiteren die Gesamtpixelanzahl  $n$  als Vergleichsgröße verwendet wird. Somit wäre eine Laufzeit von  $\mathcal{O}(n)$  zu erwarten.

---

Abbildung 6:  $7680 \times 4320$  Bild von Stachel-Polygonen[9]

Um dies zu überprüfen und den Performanzgewinn durch die SIMD-Optimierungen zu bewerten, wurde ein Benchmark mit verschiedenen großen Testbildern durchgeführt.

Iterationen	200	200	20	20	20
Bild(Pixelgröße)	Lena(262.144)	Spiral(921.600)	Circle(8.294.400)	Jimmi(22.874.400)	Poly(33.177.600)
Messung 1	9,531	33,81	30,37	81,04	114,0
Messung 2	9,531	32,40	30,35	81,04	114,1
Messung 3	9,530	33,81	31,63	81,05	114,1

Tabelle 1: Laufzeiten der Standardimplementierung bei 200 bzw. 20 Ausführungen des Laplace-Filters über verschiedenen Testbildern in Sekunden

Iterationen	200	200	20	20	20
Bild(Pixelgröße)	Lena(262.144)	Spiral(921.600)	Circle(8.294.400)	Jimmi(22.874.400)	Poly(33.177.600)
Messung 1	1,661	6,277	5,536	15,51	22,29
Messung 2	1,661	5,936	5,511	15,51	22,40
Messung 3	1,660	6,279	5,811	15,51	22,39

Tabelle 2: Laufzeiten der SIMD-Implementierung bei 200 bzw. 20 Ausführungen des Laplace-Filters über verschiedenen Testbildern in Sekunden

Die Laufzeit von  $\mathcal{O}(n)$  lässt sich gut erkennen, da der Faktor um den sich zwei Eingabegrößen unterscheiden nahezu gleich dem Faktor um den sich die Laufzeiten unterscheiden ist. Beispielsweise ist das Poly-Bild vier mal größer als das Circle-Bild und die

Laufzeit bei der ersten SIMD-Messung  $\approx 4,026$  mal länger.

Bild(Pixelgröße)	Lena(262.144)	Spiral(921.600)	Circle(8.294.400)	Jimmi(22.874.400)	Poly(33.177.600)
Messung 1	5,738	5,386	5,486	5,225	5,114
Messung 2	5,738	5,458	5,507	5,225	5,094
Messung 3	5,741	5,385	5,443	5,226	5,096

Tabelle 3: Laufzeitengewinn der SIMD-Implementierung

Vergleicht man die SIMD-Implementierung mit der normalen, so stellt man insgesamt etwa eine ca. 5 bis 6-fache Verbesserung fest. Jedoch scheint diese mit größer werdender Eingabegröße abzunehmen. Dafür gibt es jedoch keine in der Implementierung begründete offensichtliche Erklärung. Vermutlich ist dies auf Messungenauigkeiten oder Hardware/Systemeinflüsse beim Messen zurückzuführen. Dies ist eine deutliche Verbesserung und ein insgesamt zufriedenstellendes Ergebnis. Leider war es wie oben erwähnt nicht möglich eine vergleichbare und nachvollziehbare Implementierung zu finden, die obige Faltungsmatrix<sup>(1)</sup> und dabei keine anderen Filter und Bibliotheken verwendet. Beispielsweise ließen sich einige Implementierungen in anderen Sprachen wie Python oder C# finden, diese verwendeten aber während der Hauptberechnungsschleife IO-Operationen, meist mit Bildklassen aus Bibliotheken, wodurch ein aussagekräftiger Benchmark unmöglich gewesen wäre. Somit war ein Vergleich mit einer anderen Implementierung nicht möglich.

## 5 Zusammenfassung und Ausblick

Zusammenfassend, lässt sich sagen, dass die Optimierung des Laplace-Filters mittels einer SIMD-Implementierung sehr gut möglich ist. So lassen sich selbst die größten Bilder in meist unter einer Sekunde filtern (Das Poly-Bild hat eine Auflösung von  $7680 \times 4320$ ) bearbeiten. Dies sollte für die meisten Anwendungsfälle eindeutig ausreichen. Ein weiterer Aspekt des Laplace-Filters, der im Rahmen des Projekts auffiel, ist dass er, zumindest mit der verwendeten Faltungsmatrix, nicht für alle Bildtypen geeignet ist. So liefern neblige Bilder oder solche mit uneindeutigen Kanten nur sehr schlechte, nahezu vollkommen schwarze Ergebnisse.



Abbildung 7:  $7952 \times 5304$  Bild eines Stück Fleisch ist nach dem Laplace-Filter kaum zu erkennen[11]

Insofern wäre es potenziell interessant größere/genauere Faltungsmatrizen zu verwenden. Da sich dann die Berechnung wesentlich aufwendiger gestaltet, wäre auch die starke Optimierung relevanter.

## Literatur

- [1] ARM Limited. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. ARM Limited, March 2015. [https://static.docs.arm.com/den0024/a/DEN0024A\\_v8\\_architecture\\_PG.pdf](https://static.docs.arm.com/den0024/a/DEN0024A_v8_architecture_PG.pdf), visited 2020-01-27.
  - [2] ARM Limited. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. ARM Limited, December 2017. [https://static.docs.arm.com/ddi0487/ca/DDI0487C\\_a\\_armv8\\_arm.pdf](https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf), visited 2020-01-24.
  - [3] Lehrstuhl für Rechnerarchitektur und Parallelle Systeme. Lapace-filter(a200), projekt aufgabenbereich bildverarbeitung.
  - [4] ARM Holdings. A-profile architectures. <https://developer.arm.com/architectures/cpu-architecture/a-profile>, visited 2020-01-24.
  - [5] Laszlo Magyar. Interstellar, April 2016. first found on: <http://wallpapers.net/blue-and-purple-circles-hd-wallpaper/3840x2160>, visited 2020-01-27, then traced to: <https://www.deviantart.com/lacza/art/Interstellar-605924999>, visited 2020-01-27, Homepage: <http://laczadesign.com/>, visited 2020-01-27.
  - [6] Microsoft Corporation. *Bitmap Structures*. Microsoft Corporation, May 2018. <https://docs.microsoft.com/de-de/windows/win32/gdi/bitmaps>, visited 2020-01-24, Header specific: [https://docs.microsoft.com/de-de/windows/win32/gdi\(bitmap-structures?redirectedfrom=MSDN](https://docs.microsoft.com/de-de/windows/win32/gdi(bitmap-structures?redirectedfrom=MSDN), visited 2020-01-24.
-

- [7] Raspberry Pi Foundation. <https://www.raspberrypi.org/>, visited 2020-01-24.
  - [8] The GIMP Team. Gimp homepage. <https://www.gimp.org/>, visited 2020-01-25.
  - [9] unknown. 3d polygonal spikes art. first found on: <https://www.xtrafondos.com/en/wallpaper/7680x4320/3577-arte-de-picos-poligonales-neon-3d.html>, visited 2020-01-27, origin dubious, website claims all Images to be free to use.
  - [10] unknown. Jimmi hendrix collage. first found on: <https://wall.alphacoders.com/big.php?i=172618>, visited 2020-01-27, origin dubious. Seems to have been sold as Poster by multiple sources. see: [https://www.etsy.com/de/listing/711271484/collage-pop-art-jimi-hendrix-druck-auf?ga\\_order=most\\_relevant&ga\\_search\\_type=all&ga\\_view\\_type=gallery&ga\\_search\\_query=jimi+hendrix+painting&ref=sc\\_gallery-1-2&plkey=21b690f002cc80db0346b6793344a3943102d160%3A711271484&pro=1&frs=1](https://www.etsy.com/de/listing/711271484/collage-pop-art-jimi-hendrix-druck-auf?ga_order=most_relevant&ga_search_type=all&ga_view_type=gallery&ga_search_query=jimi+hendrix+painting&ref=sc_gallery-1-2&plkey=21b690f002cc80db0346b6793344a3943102d160%3A711271484&pro=1&frs=1), visited 2020-01-27 and [https://www.amazon.in/Posterhouzz-Hendrix-Singers-Poster\\_-MUZ3100/dp/B06VWL2LNR](https://www.amazon.in/Posterhouzz-Hendrix-Singers-Poster_-MUZ3100/dp/B06VWL2LNR), visited 2020-01-27.
  - [11] unknown. Super high-resolution sous vide sirloin steak. unclear origin, picture appears on various websites, seems to have first surafce on <https://imgur.com/gallery/9qtVISH>, visited 2020-01-27, posted by user stainless13 on 2015-12-08 and a day later on [https://www.reddit.com/r/FoodPorn/comments/3vvpja/super\\_highresolution\\_sous\\_vide\\_sirloin\\_steak/](https://www.reddit.com/r/FoodPorn/comments/3vvpja/super_highresolution_sous_vide_sirloin_steak/), visited 2020-01-27.
  - [12] AA VFX. 4k purple spiral waves 2160p motion background aa vfx, December 2016. [https://www.youtube.com/watch?v=-u0Pkfb\\_C-o](https://www.youtube.com/watch?v=-u0Pkfb_C-o), visited 2020-01-27, Homepage: <http://director-editor.co.il/>, visited 2020-01-27.
-