Barcodes

Seminar “Wissenschaftliches Rechnen”, ZHAW

20. Juni 2013

Florian Lüthi (luethifl@students.zhaw.ch)

https://github.com/foyan/ColorizedBarCodec

Generation 1

Unified Product Code (eindimensional)

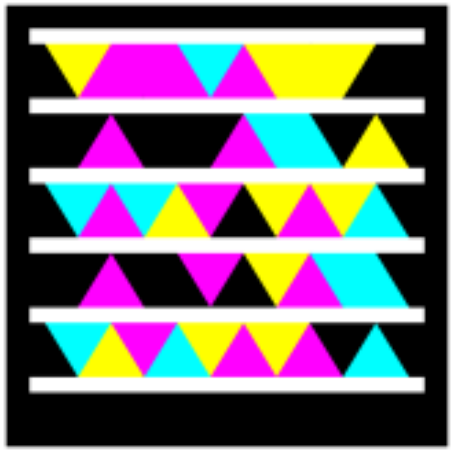
Kapazität: < 5 Bytes

Generation 2

Quick Response Code (zweidimensional)

Kapazität: 2953 Bytes

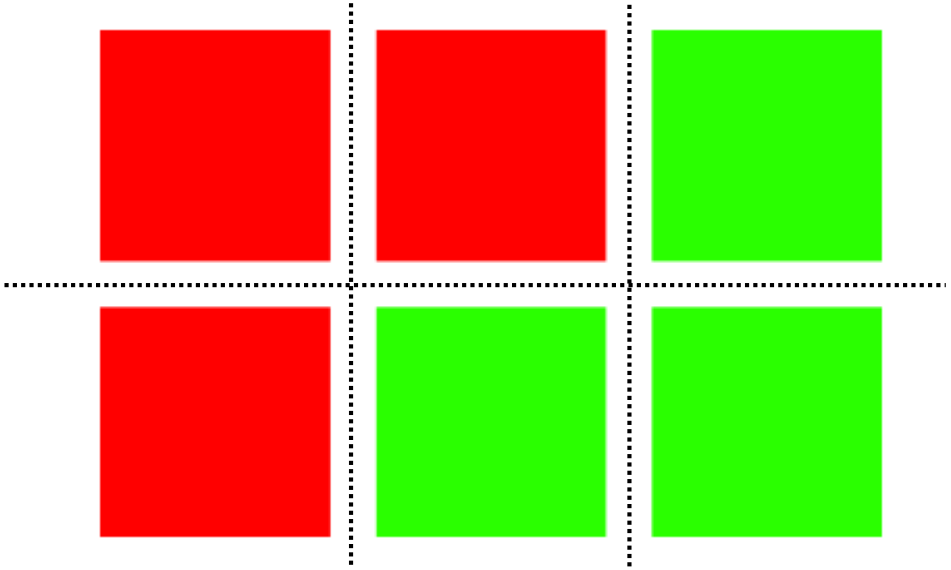
Generation 3



High Capacity Color Barcode (2D+Farben)

(Empirische) Kapazität: 2000 Bytes / 1Cent-Münze

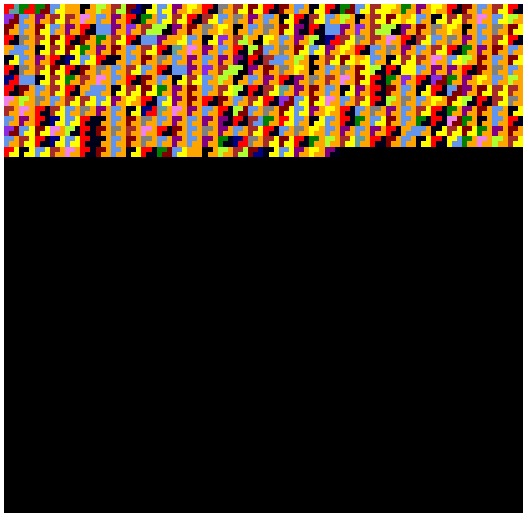
Methode von Rana et al. (I)



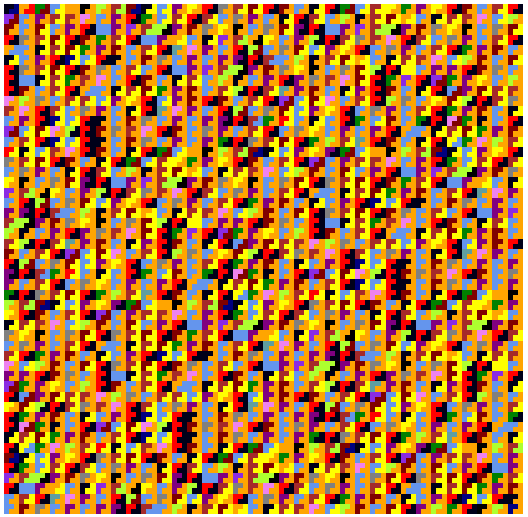
1 Byte: **0000 0001** oder **0001 0000**

Methode von Rana et al. (II)

Leerer Barcode



Was ist Aufklärung?



Was ist Aufklärung? n-mal

Methode von Rana et al. (III)

Vorteile

- Sehr einfach umsetzbar
- Lineare Zeitkomplexität

Caveats

- Immer worst-case-Optimierung
- Nicht vollständig definiert
- Keine Error-Detection
- Kapazität: 1698 Bytes

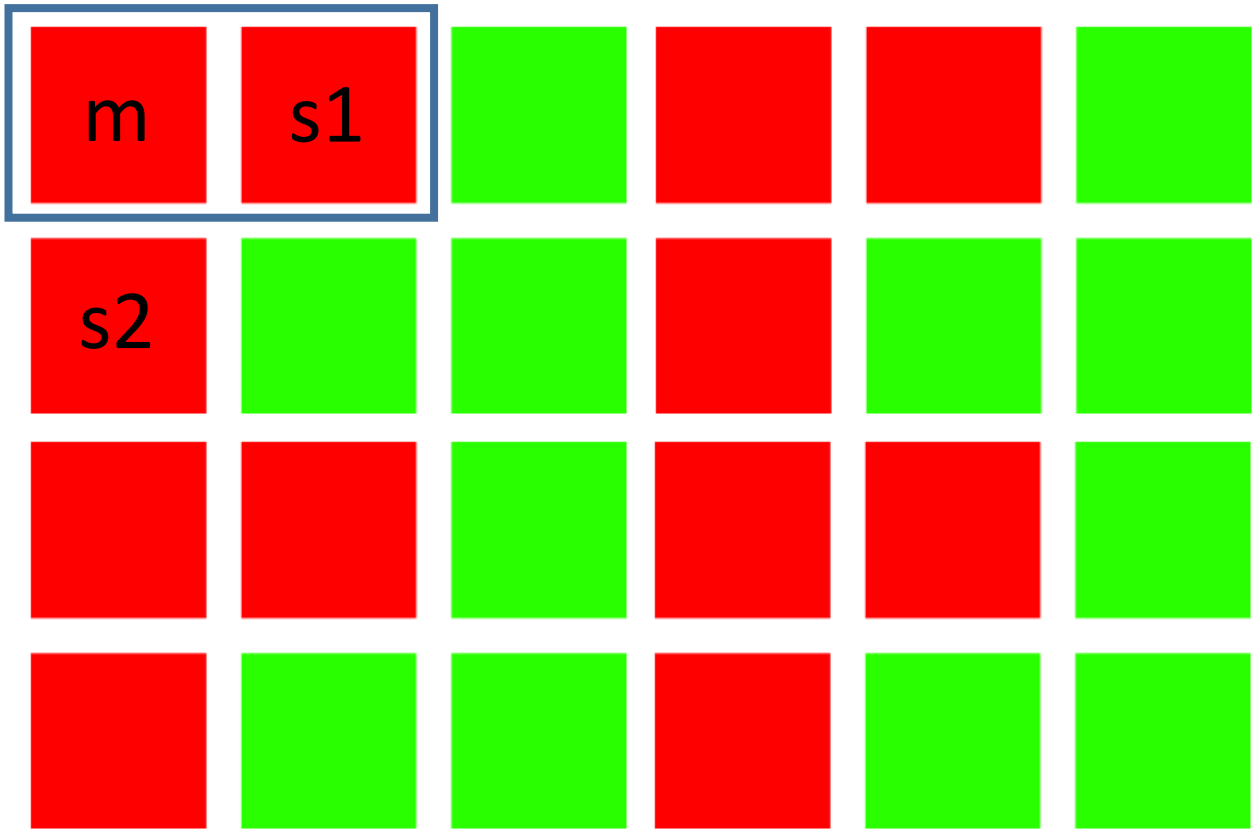
Codier-Algorithmus

```
algo encode:
        for i in [0..|text|-1]:
        var x = (i*3) mod width
        var y = (i*3) / width * 2
        bc[x, y] = bc[x+1, y] = bc[x, y+1] =
                color(text[i].low)
        bc[x+1, y+1] = bc[x+2, y+1] = bc[x+2, y] =
                color(text[i].high)
```

Decodier-Algorithmus

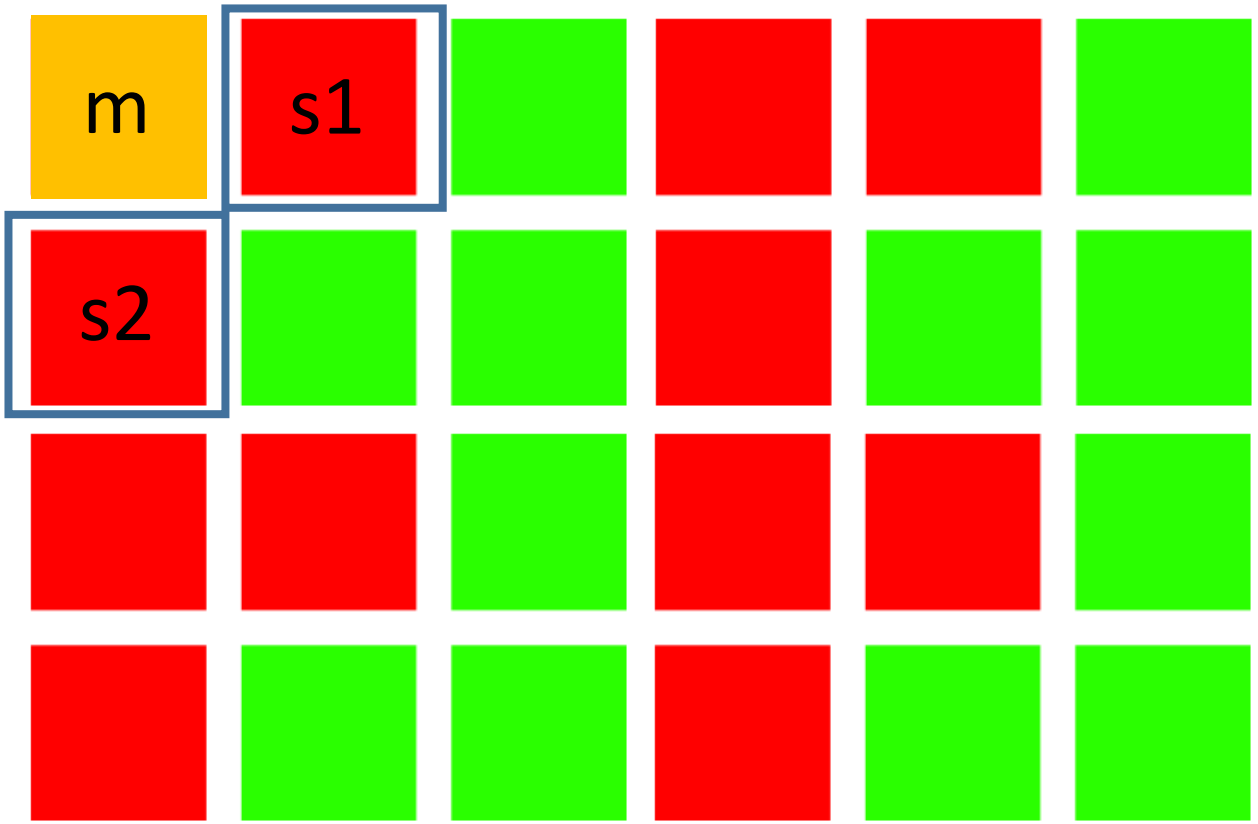
```
algo decode:
        for (y, x) in [0..height-1] x [0..width-1] step(2, 3):
                var c1 = detect_color
                        (bc[x, y], bc[x+1, y], bc[x, y+1])
                var c2 = detect_color
                        (bc[x+1, y+1], bc[x+2, y+1], bc[x+2, y])
                text.append(nibble(c2) nibble(c1))
```

“Probabilistische Fehlererkennung”



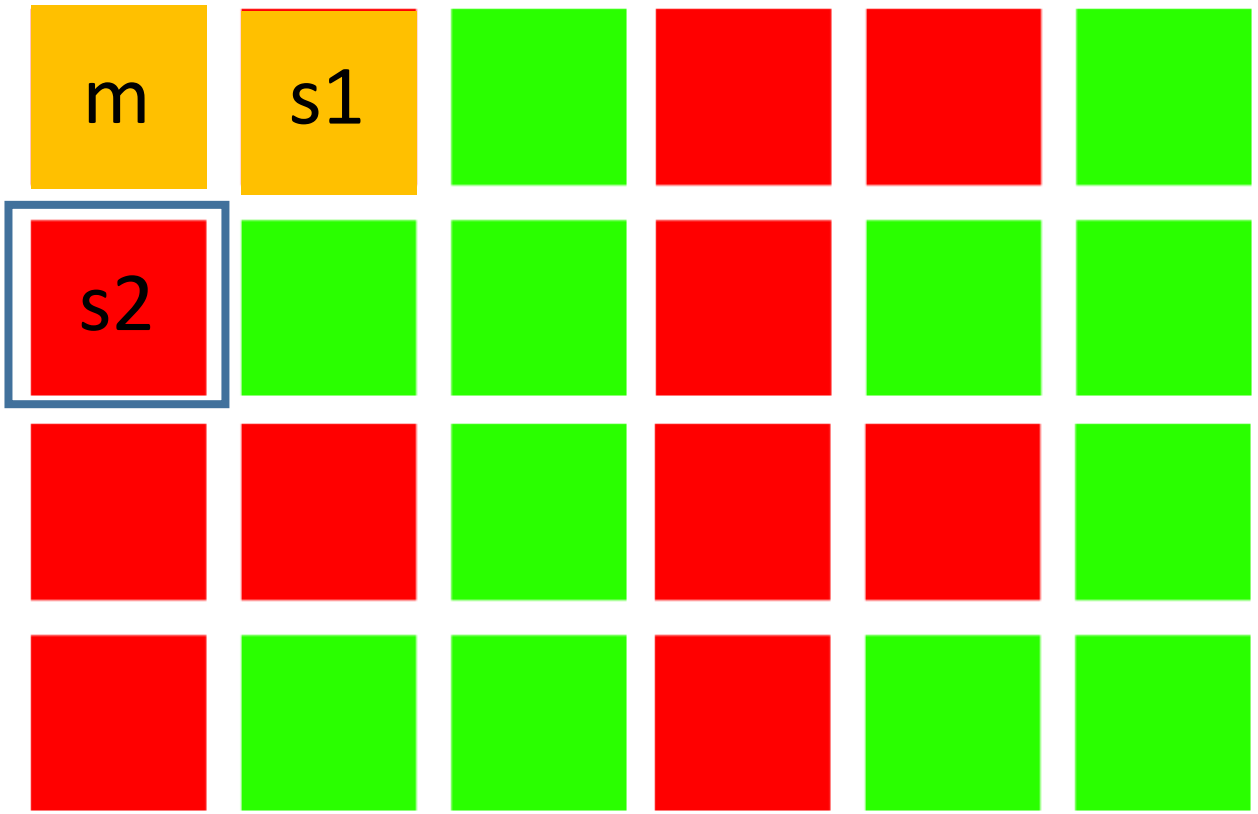
m = s1 => return m

“Probabilistische Fehlererkennung”



s1 = s2 => return s1

“Probabilistische Fehlererkennung”



return s2

Parallelisierung – mpi_task_runner

```
algo.init()          // rot => void*
for each slave node:
      var in = get_sliced_input(slave)
      invoke_slave(in)
      var my_in = get_my_input()
algo.process_slice(my_in)
result += algo.collect_slice(my_in)
for each slave node:
      result += algo.collect_slice(slave)
algo.finalize(result)
```

Parallelisierung – encoder (Bsp.)

```
void* encoder::get_sliced_input(int sl_idx, int sl_cnt) {
    string* s = new string("");
    for (unsigned int i = sl_idx; i < this->_str.length(); i += sl_cnt) {
        (*s) += this->_str[i];
    }
    return s;
}
```

mpi_task_runner

sl_idx = [0..world.size()-1]

sl_cnt = world.size()

simple_task_runner

sl_idx = 0

sl_cnt = 1

Parallelisierung – Transport (I)

Problem:

Serialisierung von **void*** per se nicht möglich

Best Practice:

Registrierte Well Known Types (gemeinsame Superklasse)

Lösung:

Custom Container **pack**

Parallelisierung – Transport (II)

```
void encoder::pack_input(pack &p, void* data) {
        p.Str = *((string*)data);
}

void encoder::pack_output(pack &p, void* data) {
        p.Pixels = *((vector<pair<rgb, rgb> >*)data);
}

void* encoder::unpack_input(pack &p) {
        return new string(p.Str);
}

void* encoder::unpack_output(pack &p) {
        return new vector<pair<rgb, rgb> >(p.Pixels);
}
```

Bidirektionale Mappings

colorizer:

```
typedef boost::bimap<unsigned short, rgb> nibble_bimap;
```

rgb:

```
template<> struct less<rgb> : binary_function<rgb, rgb, bool> {
        bool operator() (const rgb& a, const rgb& b) const {
                if (a.r() < b.r()) { return true; }
                if (a.r() > b.r()) { return false; }
                if (a.g() < b.g()) { return true; }
                if (a.g() > b.g()) { return false; }
                return a.b() < b.b();
        }
};
```

Fazit

- Verfahren funktioniert und ist umsetzbar
- Hat aber konzeptionelle Mängel
- Parallelisierbare Verarbeitung marginal gegenüber Transport-Overhead
- Threads könnten die Lösung sein