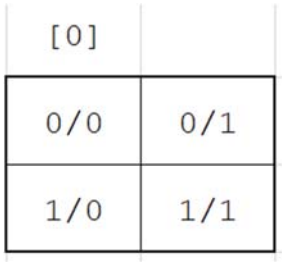
11 C-Solver

Der Solver wird in C geschrieben als CMD-Line Anwendung. Das wird gemacht da eine compiled App (binary - Maschinencode) eine schnellere Ausführungszeit hat als Python Skript Code, welcher zuerst interpretiert werden muss.

11.1 Data Representation

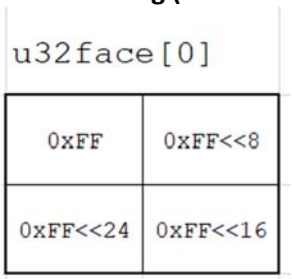
Es wird ein Facemap genutzt wie in Python.
Anstelle eines 3-Dimensionalen Arrays [6][2][2] erfolgt die Darstellung in einem u32 array der Größe [6], also in 6x4Byte. Jedes u32 repräsentiert eine Seite.

Python Darstellung



Datenreihung:
0,1
2,3

C-Darstellung (mit den Bitmasken)



Datenreihung: (byte order)
0,1
3,2

ACHTUNG:
Die Datenreihung wurde für die C-Darstellung geändert. Dadurch kann die „face rotation“ mit einem einzelnen CPU Kommando (ROTR bzw. ROTL) durchgeführt werden.
ROTR (rotate right) … rotate_face_CCW
Es ist zu prüfen wie der maschinencode aussieht, da man die rotation ja nicht direkt beschreiben kann in C.

Der Source-Code zur Rotation. u8num ist in diesem Fall 8, 16 oder 24.
Wie man sieht muss man die Rotation durch Schiebe Operation und OR realisieren

```
ROTR = ((u32face >> u8num)|((u32face << (32-u8num))));
ROTL = ((u32face << u8num)|((u32face >> (32-u8num))));
```

Ein ROTR mit 24Bits entspricht natürlich gleichermaßen einem ROTL mit 8 Bits in diesem Beispiel.

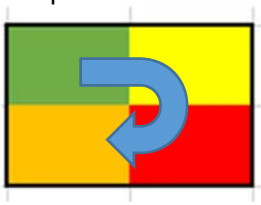
11.2 Kodierung der Farb-Information

```
enum COLOR_BITMAP{
    COL_IDX_WHITE   = 0,        //0b00000000
    COL_IDX_YELLOW  = 1,        //0b00000001
    COL_IDX_ORANGE  = 2,        //0b00000010
    COL_IDX_RED     = 3,        //0b00000011
    COL_IDX_BLUE    = 4,        //0b00000100
    COL_IDX_GREEN   = 5,        //0b00000101
};
```

Die Farbinformation in Dezimal kodierung passt in 3 Bits. Je face wird also 4*3=12Bits Information benötigt, womit man auch mit einem u16 auskommen würde für den 2x2 cube.

Es wird aber ein u32 genommen da dieser auch für den 3x3 cube passen wird (8*3 = 24Bits, der center-piece ist nicht relevant für den solver da er beim 3x3 nicht veränderbar ist)

Beispiel:



Gemäß Datenreihung:
„Green-Yellow-Red-0range“ > „5-1-3-2“

0xFF	0xFF<<8
0xFF<<24	0xFF<<16

u32face = 2<<24 | 3<<16 | 1<<8 | 5
u32face = TL (top-left) |TR | BR | BL (bot-left)

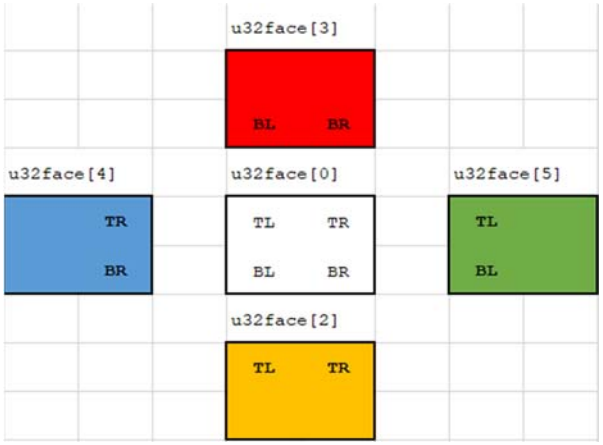
u32face = 0x 02 03 01 05 = 0b 00000**010** 00000**011** 00000**001** 00000**101**
ROTR (CCW) = 0x 05 02 03 01 = 0b 00000**101** 00000**010** 00000**011** 00000**001**
ROTL (CW) = 0x 03 01 05 02 = 0b 00000**011** 00000**001** 00000**101** 00000**010**

Hier sieht man ganz gut die ungenutzten Bits und dass es in den u16 reinpasst.

11.3 Umgang mit den Angrenzenden faces – Var 1

Austausch auf Elemente-Basis
TL…TOP LEFT
TR…TOP RIGHT
usw.

Hier sieht man die Beteiligten Datenelement bei einer Rotation der weißen Seite. Wie man sieht ist von den 4 angrenzenden Seiten bei gewählter Daten-Repräsentation die Information immer an einer anderen Position im u32face zu finden.



Für dieses Beispiel kann die rotation der angrenzenden Seite wie wie folgt beschrieben werden für ein CW rotation:
u32_mem = u32face[5]
TL_SET(u32face[5], BL_GET(u32face[3]);
BL_SET(u32face[5], BR_GET(u32face[3]);

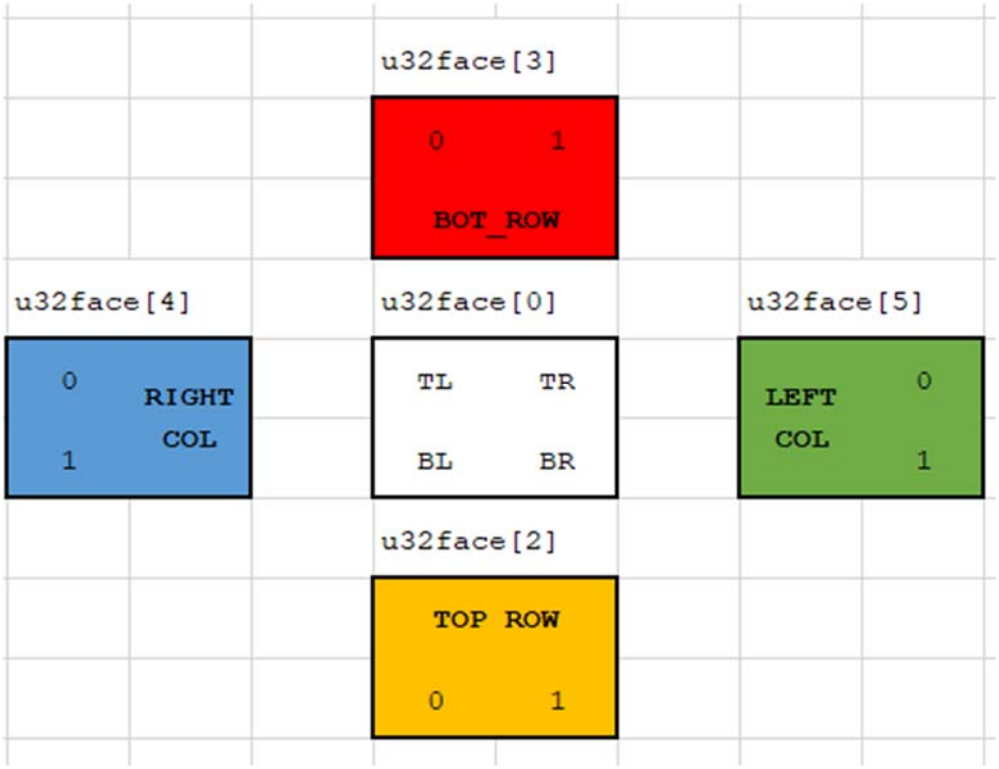
BL_SET(u32face[3], BR_GET(u32face[4]);
BR_SET(u32face[3], TR_GET(u32face[4]);
usw.

das lässt sich nicht sehr elegant umsetzen und ist kaum leserlich. Auch ist es ein großer Aufwand für die CPU, da jeweils nur 3 Bit in einer Zeile verschoben werden.

Auch die denkbare Erweiterung auf einen 3x3 / NxN cube explodiert im Aufwand des codings sowie CPU.

11.4 Umgang mit den Angrenzenden faces – Var 2

Austausch auf Zeilen / Spalten Basis



Diese Variante erscheint in der Darstellung bereits einfacher, wie man an der vollständigen Beschreibung für die CW-Rotation sieht:

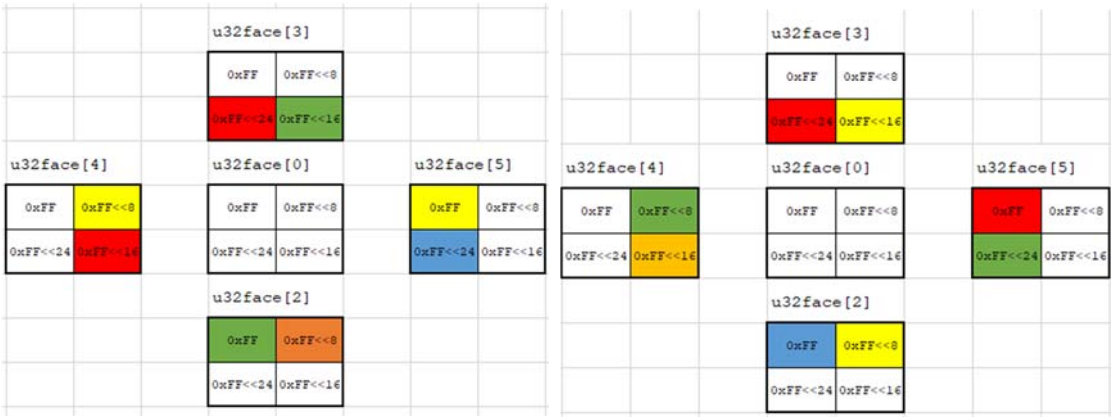
u32_mem = u32face[5]
LEFT_COL_SET(u32face[5], BOT_ROW_GET(u32face[3]));
BOT_ROW_SET(u32face[3], RIGHT_COL_GET(u32face[4]));
RIGHT_COL_SET(u32face[4], TOP_ROW_GET(u32face[2]));
TOP_ROW_SET(u32face[2], LEFT_COL_GET (u32_mem));

Es bleibt die Unschönheit dass die Rohinformation jeweils an einer anderen Bit-Position steht und nicht direkt verarbeitet werden können

11.4.1 Beispiel – Rotation „U“

Bild: VORHER

BILD: NACH ROTATE_CW



Die entsprechenden Bitmasken für Reihen & Spalten:

BOT_ROW (face[3])	MASK:	0xFFFF 0000
TOP_ROW (face[2])	MASK:	0x0000 FFFF
RIGHT_COL (face[4])	MASK:	0x00FF FF00
LEFT_COL(face[5])	MASK:	0xFF00 00FF

Und den entsprechenden Dateninhalten:

BOT_ROW (face[3])	DATA:	0x0305 0000
TOP_ROW (face[2])	DATA:	0x0000 0205
RIGHT_COL (face[4])	DATA:	0x0003 0100
LEFT_COL (face[5])	DATA:	0x0400 0001

Drehung CW – Daten-Transformation:

Face[3] > Face[5]	MASK:	0xFFFF 0000 > 0xFF00 00FF	
Face[3] > Face[5]	DATA:	0x0305 0000 > 0x0500 0003	(ROTL)
Face[5] > Face[2]	MASK:	0xFF00 00FF > 0x0000 FFFF	
Face[5] > Face[2]	DATA:	0x0400 0001 > 0x0000 0104	(ROTL)
Face[2] > Face[4]	MASK:	0x0000 FFFF > 0x00FF FF00	
Face[2] > Face[4]	DATA:	0x0000 0205 > 0x0002 0500	(ROTL)
Face[4] > Face[3]	MASK:	0x00FF FF00 > 0xFFFF 0000	
Face[4] > Face[3]	DATA:	0x0003 0100 > 0x0301 0000	(ROTL)

Damit ist der Transformations-Algorithmus wie folgt zu formulieren:

```
Face[3] > Face[5]
Face[5] &= ~LEFT_COL_MASK;                    //zu ändernde Bits löschen = 0
Face[5] |=  (LEFT_COL_MASK & ROTL(Face[3], 8));    //korrekt Bits setzen = 1
```

Selbiges lässt sich 1:1 Anwenden für ROTATE_CCW, jedoch dann mit ROTR.

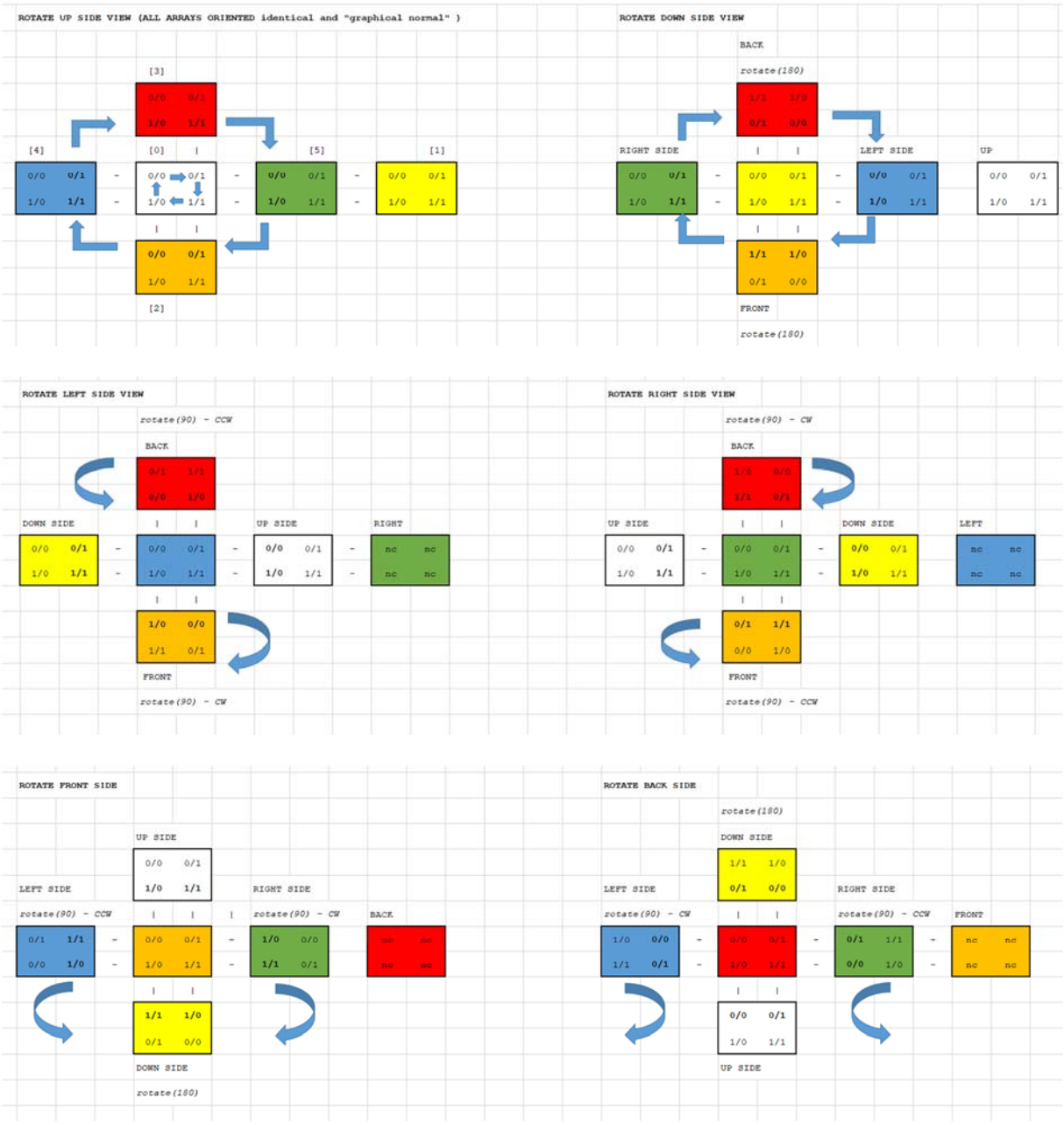
Solange das facemap einer Seite in ein integer passt, der durch CPU-instruction rotierbar ist, ist dies eine sehr elegante Lösung.

11.5 Umsetzung für ALLE Möglichen Seiten Rotationen

(Anhand der Python Umsetzung des 3x3 cube bekannt)

Es ergibt sich folgendes Schema für pre-processing Aktionen wenn man eine Seite drehen will:

UP-SIDE:	no pre-processing		
DOWN-SIDE:	BACK: 180°	FRONT: 180°	
LEFT-SIDE:	BACK: 90° CCW	FRONT: 90° CW	
RIGHT-SIDE:	BACK: 90° CW	FRONT: 90° CCW	
FRONT-SIDE:	LEFT: 90° CCW	RIGHT: 90° CW	DOWN: 180°
BACK-SIDE:	LEFT: 90° CW	RIGHT: 90° CCW	DOWN: 180°



Es braucht es also bei den Aktionen noch zusätzliche rotationen an faces, die jedoch allesamt wieder auf bitweise rotation zurückzuführen sind. Lediglich die Rotation an der UP-SIDE (U, U‘, U2) erfordert kein Preprocessing. Diese Schritte sollten die CPU-Zeit jedoch nicht verändern, da lediglich die Anzahl in der Bit-Rotation geändert wird und keine zusätzlichen Instructions hinzukommen

Im Python Skript-Code ist aktuell die Variante #1 umgesetzt mit einem remap der Einzelnen-Elemente.
Für den C-Solver soll die Variant #2 umgesetzt werden.

Die Python Implementierung kann dabei als Test-Environment genutzt werden zur Validierung der C-Implementation.