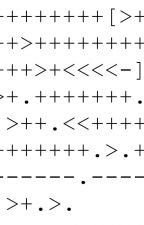
Grafik med Brainfuck



Navn: Simon Lykke Andersen



Fag: Programmering C

Lærer: Anders Juul Refslund Petersen

Afleveringsdato: 5/4-2020

Afleveringssted: H.C. Ørsted Lyngby HTX, Lectio

Indholdsfortegnelse

[Indholdsfortegelse 2](#_Toc36823116)

[Synopsis 3](#_Toc36823117)

[Indledning 3](#_Toc36823118)

[Projektformulering/problemformulering og kravspecifikation 4](#_Toc36823119)

[Udviklingsmiljø 4](#_Toc36823120)

[Visual Studio Code 4](#_Toc36823121)

[Chrome 5](#_Toc36823122)

[Runtime beskrivelse 6](#_Toc36823123)

[Implementering 9](#_Toc36823124)

[Bagvedliggende del 9](#_Toc36823125)

[Overblik over klasserne 9](#_Toc36823126)

[Step metode 10](#_Toc36823127)

[Stack 11](#_Toc36823128)

[Brugergrænseflade 11](#_Toc36823129)

[HTML 11](#_Toc36823130)

[JavaScript 13](#_Toc36823131)

[CSS 13](#_Toc36823132)

[Program test 15](#_Toc36823133)

[Konklusion 15](#_Toc36823134)

[Bilag 16](#_Toc36823135)

[Bilag 1 - index.html 16](#_Toc36823136)

[Bilag 2 - bf.js 25](#_Toc36823137)

[Bilag 3 - Flowchart over Step metode fra InterpreterWithStack 31](#_Toc36823138)

[Bilag 4 - Prædefinerede processer til flowchart over Step metode fra InterpreterWithStack 32](#_Toc36823139)

# Synopsis

Brainfuck er et af de mest kendte esoteriske programmeringssprog. Esoteriske programmeringssprog er programmeringssprog, der er lavet for at være udfordrende, sjovt, interessant og anderledes at programmer i. Brainfuck kode består af 8 symboler (< > + - , . [ ]), derudover er der et bånd med ”uendelig” mange positive 8-bit heltal og en pointer til et bestemt sted på dette bånd. Symbolerne giver mulighed for at flytte på pointeren, manipulere båndet, input og output og de firkantede parenteser er en slags loop, der kører hvis værdien pointeren peger på ikke er nul. Mulighederne for hvilke slags programmer man kan lave med Brainfuck er ret begrænsede da man kun kan tage tekst som input og give tekst output. Jeg ville derfor lave en måde hvor man kunne bruge nogle af funktionerne fra JavaScript biblioteket p5.js, hvilket er et JavaScript bibliotek, der gør det muligt at tegne og interagerer med et canvas. For at gøre dette implementerede jeg en Brainfuck fortolker, der kunne eksekvere selve Brainfuck koden. Derefter integrerede jeg p5’s funktioner ved at funktionerne var repræsenteret som et tal (jeg har valgt at kalde opcode, kort for operation code). Denne opcode skubber man til en stack, der eksisterer separat fra det øvrige Brainfuck økosystem, man skubber også argumenterne til disse funktioner på stakken. Jeg har udvidet Brainfucks spartanske symbolsæt med ”:” og ”;”, der henholdsvis er skub opcode og skub argument (i koden pushOPC og pushARG).

Mit projekt kan findes her: <https://simon220902.github.io/BrainFuckProcessing/>

Kildekoden kan findes her: <https://github.com/Simon220902/BrainFuckProcessing>

# Indledning

Inden jeg begyndte på dette projekt, overvejede jeg nøje hvilke biblioteker og hvilket programmeringssprog jeg skulle anvende. Jeg vidste at jeg skulle anvende en eller anden form for bibliotek, der kunne rendere grafik da det var en af grundstenene i mit projekt. Jeg overvejede at anvende processing, da det var det vi havde anvendt i den hidtidige undervisning. Processing ville fungere fint til at udføre selve grafikdelen af mit program, men jeg ville også gerne have en intuitiv GUI til selve eksekveringen af mit program. Der findes biblioteker i processing, der hjælper til at kunne lave flotte GUI’er, men så skulle jeg også kunne lave endnu et vindue fra samme program, hvilket i mine undersøgelser så nært umuligt ud. Næste mulighed jeg overvejede, var at lave det i python med pygame som den grafiske del. Jeg har selv arbejdet meget med python og pygame biblioteket, bl.a. til en TuringMaskine, hvor der var et tilhørende kommandolinje interface med mulighed for grafisk eksekvering i pygame. Dog ville dette kræve mere for at kunne køre da den mulige bruger skulle have python og pygame installeret. Sidste mulighed, hvilket var den jeg endte med at vælge, var at bruge p5.js biblioteket til Brainfucks grafiske del og almindelig JavaScript til selve Brainfuck fortolkeren. Denne mulighed gjorde det muligt at bruge HTML til at lave GUI’en til selve eksekveringen.

# Projektformulering/problemformulering og kravspecifikation

Projektformulering:

Lav en version af Brainfuck, der gør det muligt at lave programmer, der kan producere grafiske output, helst med mulighed for brugerinteraktion, i Brainfuck.

For at være sikker på at mit program i slutningen af dette projekt opnåede dette havde jeg defineret en række krav, der videre specificerede hvad det rent faktisk skulle til for at opnå målet.

Kravspecifikation:

* Fungerende Brainfuck fortolker
* Mulighed for at kalde tegne funktioner i Brainfuck
* Mulighed for at få bruger input i Brainfuck (keyboard og mus)
* Miljø til at kunne eksekvere koden i (med grafisk display)
* Miljø til at kunne gå trinvist gennem koden

# Udviklingsmiljø

I dette projekt har jeg anvendt Visual Studio Code som editor og hvor jeg løbende har testet min kode i Chrome, hvor jeg har anvendt deres udviklerværktøjer til at undersøge sidens elementer.

## Visual Studio Code

Et billede, der indeholder skærmbillede, monitor

Automatisk genereret beskrivelse

Figur 1: Overblik over Visual Studio Code

Visual Studio Code er en kode editor udviklet af Microsoft, hvor det er muligt at downloade en lang række udvidelser, selvfølgelig syntax highlighting af en lang række sprog, men også mere komplekse ting. I dette projekt anvendte jeg det bl.a. til se en renderet version af min README.md, hvilket er det man på GitHub bruger til at fortælle om sit projekt, hvilket jeg i starten brugte som en notesbog over projektet.

## Chrome

Et billede, der indeholder skærmbillede

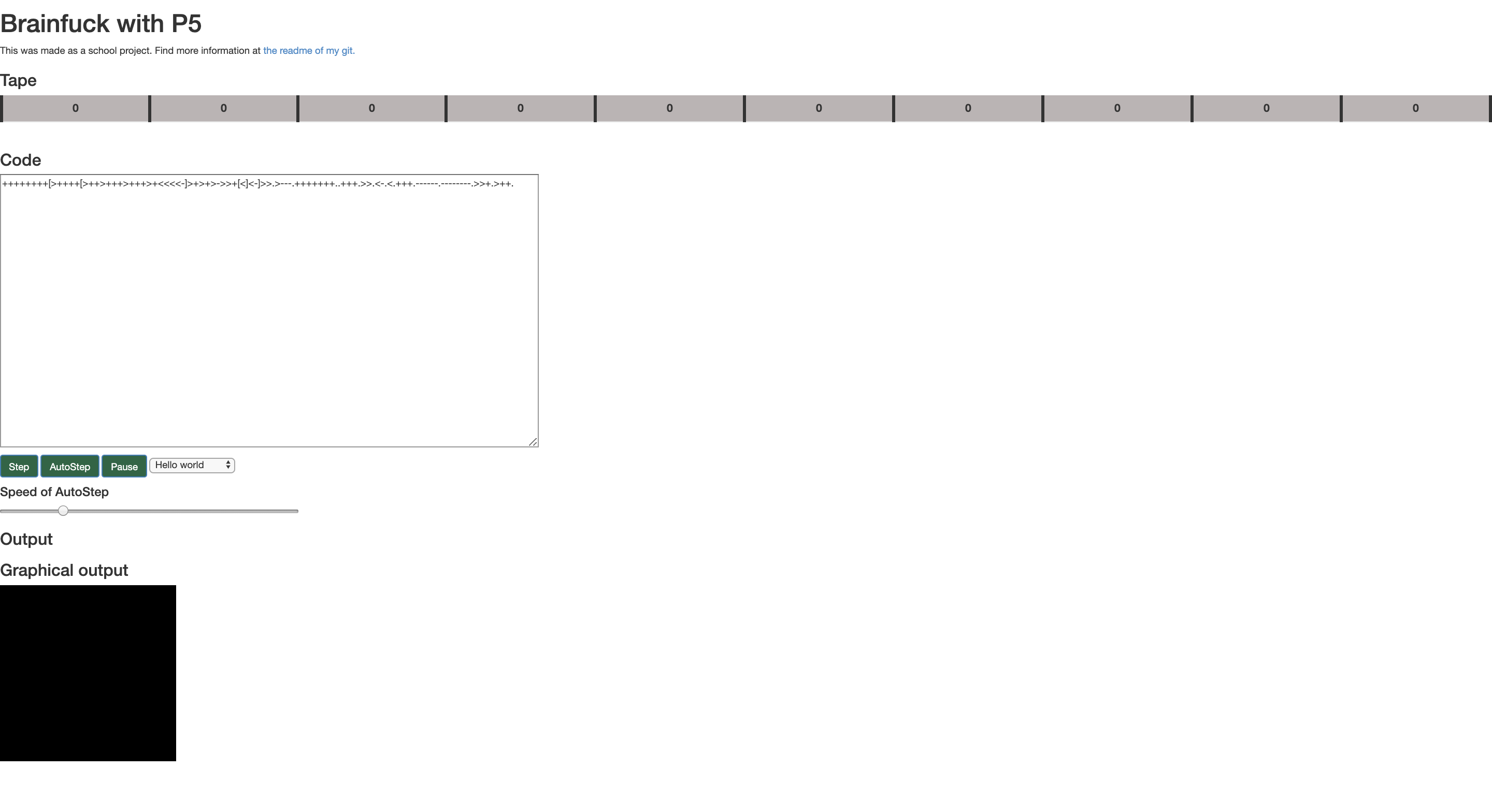
Automatisk genereret beskrivelse

Figur 2: Overblik over Google Chrome med udviklingsværktøjer åben.

Chrome er en browser udviklet af Google og er en af de mest anvendte browsere i verden. Chrome er også en af de browsere med de bedste udviklingsværktøjer. Disse udviklingsværktøjer er bl.a. et værktøj, der gør det muligt at kunne finde den specifikke HTML kode, der står bag et element på en hjemmeside, ved bare at klikke/holde musen over dette element. Derudover er der en konsol, hvor JavaScript fejlmeddelelser kommer op. Denne konsol har jeg blandt andet også anvendt til traditionel debugging ved at udskrive værdier forskellige steder i koden ved JavaScript kommandoen ”console.log(…)”.

# Runtime beskrivelse

Nedenunder ses selve brugergrænsefladen til fortolkeren. Jeg har prøvet at holde siden simpel for at gøre den så overskuelig som mulig. Jeg vil nedenunder gå igennem de forskellige delelementer.



Figur 3: Overblik over hele programmets brugergrænseflade

Øverst på siden er der en lang bar, hvor der står en række tal. Dette er Brainfucks memory den såkaldte det jeg i synopsen kaldte båndet, hvilket jeg herfra vil kalde tape. Når man kører et program, bliver pladsen (mere specifikt den grå baggrund) på tapen man befinder sig på rød. Som ses på billederne nedenunder. Hvis pladsen der peges på, er udenfor de ti (der er lavet) sættes pladsen som værende i midten, hvor tallene så bare skifter omkring pladsen.



Figur 4: Tapen i start position hvor alle værdier sat til nul



Figur 5: Tapen efter at være blevet redigeret på, hvor vi ser max værdien i det første felt og det felt der peges på er markeret med rød.

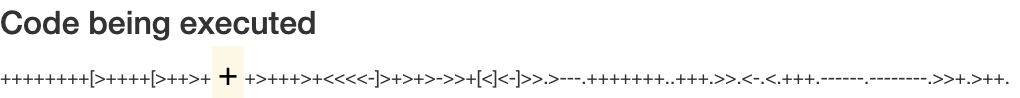
Nedunder tapen er et tekstvindue, hvor brugeren kan redigere koden.

Et billede, der indeholder skærmbillede

Automatisk genereret beskrivelse

Figur 6: Kode vinduet loadet med BF’s Hello World program.

Under tekstvinduet er et afsnit, der under eksekvering, får samme tekst som i tekstvinduet ovenover, men det symbol, der eksekveres i det øjeblik, bliver highlightet og gjort større.



Figur 7: Eksempel på highlighting af det eksekverede symbol.

Under afsnittet med det highligtede symbol er en række knapper, en dropdown menu og en slider. Den første knap ”Step” kører et enkelt step, den eksekverer altså et symbol. Den anden knap ”AutoStep” gør sådan at Step funktionen kaldes ved et fast interval som bestemmes ved værdien af slideren. Den tredje knap ”Pause” sætter programmet på pause, dette er kun nødvendigt ved AutoStep. Herefter har vi en dropdownmenu med en række eksempel programmer, hvor ”Hus” programmet er valgt som default. Det er kun programmet ”Cirkler ved mus” og ”Hus” som anvender det grafiske output. Slideren bestemmer tiden mellem hver gang AutoStep bliver kaldt, hvor tallet er ms mellem Step funktionen bliver kaldt. Man skal dog pause programmet for at ændringen i ms bliver anvendt.

Et billede, der indeholder skærmbillede

Automatisk genereret beskrivelse

Figur 8: Knapper og andre slags input til styring af eksekveringen af koden.

Et billede, der indeholder skilt, sidder, monitor, telefon

Automatisk genereret beskrivelse

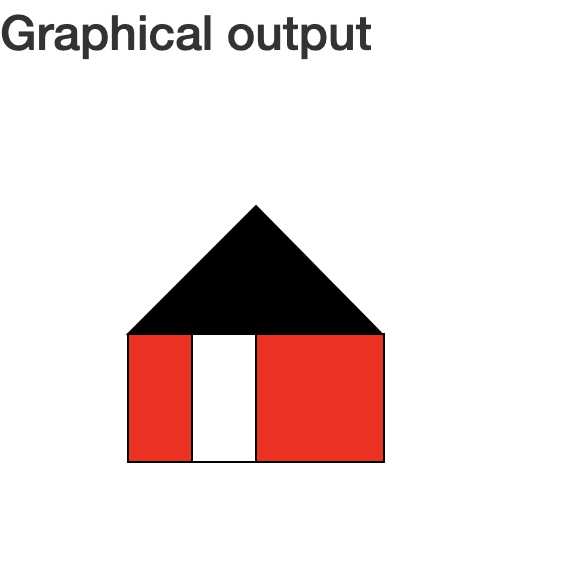
Figur 9: Dropdownmenuen foldet ud.

Under knapperne m.m. er tekst outputtet som et afsnit. Her er hvordan outputtet ser ud efter ”Hello World” programmet har kørt.



Figur 10: Output feltet, hvor programmets tekst output skrives.

Under tekst outputtet er det grafiske output. Dette er vinduet, hvor udfaldet af de grafiske funktioner kaldt fra Brainfuck vil kunne ses. Her er vinduet efter programmet ”Hus” har kørt.



Figur 11: Grafik vinduet, hvor programmets grafik vises og kan interageres med.

# Implementering

Overordnet set består mit program af to dele. Den ene del er det man ser på selve siden altså GUI’en/brugergrænsefladen til eksekveringen af Brainfuck koden, den anden del er den bagvedliggende del, der gør brugergrænsefladen mulig altså selve fortolkeren. Jeg vil inkludere det grafiske tillæg til Brainfuck i den anden del selvom det i koden står sammen med brugergrænsefladens. Jeg vil starte med at beskrive den bagvedliggende del.

## Bagvedliggende del

### Overblik over klasserne

Delen hvor fortolkeren og den grafiske del af Brainfuck er lavet består af en række klasser, objekter og interaktioner mellem disse. Dette har jeg beskrevet ved et klasse diagram, der ses nedenunder.

Et billede, der indeholder skærmbillede

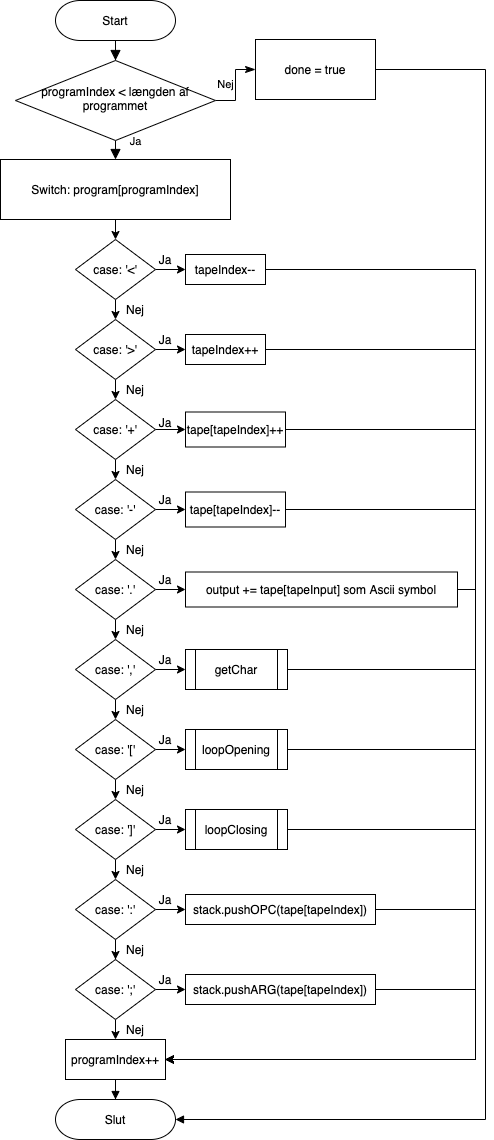
Automatisk genereret beskrivelse

Figur 12: Klassediagram over de anvendte klasser, med anvendelses pile.

Dette giver et overblik over hvordan de forskellige bestanddele interagerer med hinanden. Jeg har forsøgt at anvende UML’s konventioner omkring pilenes form. Men jeg vil lige gennemgå hvad diagrammet rent faktisk betyder. Vi har en klasse Interpreter som er selve Brainfuck fortolkeren, men uden muligheden for at anvende de grafiske funktioner. Denne klasse har to metoder, step og eval. Step funktionen kører kun et step (altså håndtere et symbol), hvor eval kører til programmet er færdigt. Så er der en klasse InterpreterWithStack som er nedarvet fra vores Interpreter klasse. Denne klasse har endnu et medlem med navnet stack som er en instans af Stack klassen (som jeg vil forklare mere om senere). Den har også to metoder (med samme navn som superklassen), men som er redefineret for at kunne håndtere to ekstra symboler nemlig ”:” og ”;”, hvilket er dem der fra Brainfuck interagerer med det grafiske (altså stakken). Så er der så Stack klassen, stack klassen er den som står for den grafiske del, den kalder nemlig p5 funktionerne når den nødvendige information er der. Stack klassen indeholder en instans af den interpreter, den selv er en instans hos. Det gør den fordi der er nogle opcodes, der ”returnerer” en værdi som bare bliver lagt på den plads på tapen fortolkeren er lige nu. Derudover har den et andet medlem med navnet stack som er det, der indeholder de kaldte opcodes og de tilhørende argumenter. Så er der de to metoder pushOPC og pushARG, der står for at sætte de nødvendige informationer på stakken og at eksekvere tegnefunktionerne når det nødvendige antal argumenter er givet. Til sidst er der klassen p5, hvor der anvendes en instans af p5 til at kunne tegne med. Da p5 klassen har super mange medlemsvariabler og metoder har jeg valgt ikke at specificere dem yderligere, vi anvender kun dem, der kaldes direkte fra Stack klassen.

### Step metode

Med dette generelle overblik over hvordan de forskellige klasser interagerer vil jeg nu vise en af de mest anvendte metoder, nemlig InterpreterWithStack’ens step metode. Jeg har lavet et flowchart over Step metoden. Step metoden eksekverer et symbol af programmet. I dette flowchart anvender jeg en del prædefinerede processer, disses flowcharts kan alle findes i bilag 4 sammen med en større version af flowchartet nedenunder i bilag 3. I step metoden anvendte jeg et switch case, hvilket var en smule udfordrende at afbildede i flowchartet, da switch caset i virkeligheden laver et hashtable over de mulige værdier, i stedet for at gå dem slavisk igennem som var det en række if-else if… statements som jeg har afbilledet det som.



Figur 13: Flowchart over InterpreterWithStack’s Step metode.

### Stack

I ovenstående flowchart er det eneste, der ikke laver om på klassens egen værdier, er når skubbes ting på stakken. Jeg vil nu forklare hvad denne stack er og hvordan den fungerer. Som også sås i UML diagrammet har jeg lavet en Stack klasse. Jeg vil først påpege at denne Stack klasse er ikke det samme som forstås ved en normal stack, der er en veletableret datatype, men mere en specifik anvendelse af sådan en. I moderne højniveau programmeringssprog er der ikke en separat stack datatype, men derimod metoderne push og pop, man kan anvende på lister for at få samme resultat. Dette er sådan en liste, der er gemt i Stakkens stack medlemsvariabel. I denne liste skubber vi opCodesne med en tilhørende liste til argumenter. Hvis der skubbes et argument, sættes det ind i det øverste elements argumentliste. Hver gang der skubbes en opCode eller et argument tjekkes der om det nødvendige antal argumenter er tilstede. Hvis det nødvendige antal argumenter er tilstede eksekverer den opCodens tilhørende funktion med de skubbede argumenter. Derefter popper den opCoden med argumentlisten af stakken. Nedenunder ses en repræsentation af hvad der blev beskrevet ovenover.

Et billede, der indeholder skærmbillede

Automatisk genereret beskrivelse

Figur 14: Visualisering af stakken med opCode, argumenter og resultat (angivet boksen pilen peger på).

## Brugergrænseflade

Nu er vi nået til den del af programmet, der rent faktisk gør det muligt at køre sit program osv. nemlig brugergrænsefladen. Da denne brugergrænseflade er en hjemmeside i HTML, kan vi inddele brugergrænsefladekoden i tre dele. Den første del er det der står for de elementer man ser på siden, det der identificeres som traditionel HTML, den anden del er den JavaScript kode, der står for en stor del af funktionaliteten bag disse elementer og den tredje del er den, der sørger for udseendet af de elementer man ser på siden, hvilket er beskrevet i noget kaldet Stylesheets hvor internettets Stylesheet standard er CSS.

### HTML

Før jeg begyndte at skrive HTML koden fik jeg mig et overblik over hvilke brugergrænseflade elementer jeg behøvede. Herfra var de to største krav, der berørte brugergrænsefladen, henholdsvis kravet om at programmet skulle indeholde et ”Miljø til at kunne eksekvere koden i (med grafisk display)” og ”Miljø til at kunne gå trinvist gennem koden”. Selvom disse krav var relativt åbne, havde jeg en vis forventning om hvad dette indebar. Ved det første krav mente jeg et IDE lignende miljø, hvor man både kunne skrive sin kode, se eksempler, naturlig integration af Brainfucks traditionelle input-output såvel som det grafiske vindue hele mit projekt omspinder. Til det andet krav var det selvfølgelig at der i dette miljø til eksekvering skulle være en måde at gå trinvist igennem koden, men dette var dog ikke alt hvad jeg så nødvendigt. Det man ofte har brug for nå man går trinvist igennem kode er at se hvordan værdierne ændrer sig ved hver operation og se om de gør som man havde forventet. De eneste værdier der er tilgængelige i Brainfuck er tapen, hvilken jeg så valgte skulle indgå i min brugergrænseflade.

For at opnå disse krav og tanker jeg havde til min brugergrænseflade, krævede det at jeg anvendte en lang række HTML elementer. Der var selvfølgelig de klassiske tekstelementer som overskrifter (<h1></h1>) og afsnit (<p></p>), hvor nogle fra den sidste gruppe blev gjort interaktive. Udover disse anvendte jeg også et tabelelement til min tape, da der i Brainfuck kun anvendes en endimensionel tape definerede jeg kun en række med ti kolonner, hvilket skulle repræsentere min tape. Indholdet af denne tabel skulle kunne laves om af JavaScript koden, der forklare i næste sektion, derfor var det nødvendigt at give hver af kolonnerne systematiske navne (t0-t9).

Et billede, der indeholder telefon, computer, monitor, skærm

Automatisk genereret beskrivelse

Figur 15: HTML koden bag min tape.

Udover tabellen anvendte jeg også HTML elementet textarea som værende hovedinputfeltet hvor brugerens kode skulle stå og kunne redigeres i. Til at kontrollere selve eksekveringen af programmet anvendte jeg knapper, en slider og en dropdownmenu til eksemplerne. Knapperne blev lavet ved <button></button> tagget, hvor deres funktionalitet var beskrevet/kodet i JavaScript udenfor kroppen af HTML’en. I HTML er der et input element, indenfor dette input element er der en lang række forskellige slags input elementer, hvor det ønskede element specificeres i en type deklaration inden i tagget. Det specifikke input element jeg ønskede til min slider, er af en type kaldet range, hvor man så specificerer mellem hvilke værdier denne slider skal kunne være og i hvor store trin man kan ændre på dette. Da min slider skulle repræsentere tiden mellem at fortolkerens stepfunktion blev kaldt, hvilket skulle angives i millisekunder, valgte jeg at min slider skulle repræsentere tiden i m/s, hvor brugeren kunne vælge alle værdier mellem 0 m/s og 500 m/s med 1 m/s trin. Den resulterende HTML kode så således ud:

*Figur 16: HTML koden bag slideren.*

Til dropdownmenuen med eksempel programmer anvendte jeg select elementet, hvor hver af dropdownmenuens muligheder blev specificeret imellem select elementets åbnings og luknings tag med et option tag. Da disse options også skulle indeholde selve koden af eksempel programmerne er dette ikke det pæneste HTML, men herunder ses HTML koden til dropdownmenuen:

Et billede, der indeholder skærmbillede, monitor, sidder, bord

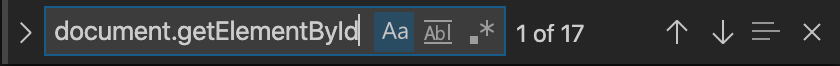
Automatisk genereret beskrivelse

*Figur 17: HTML koden bag dropwdownmenuen.*

Det sidste HTML element jeg anvendte, var til specificeringen af hvor p5.js vindue skulle være. Dette blev ved en div klasse, hvilket er en måde at organisere ting i HTML på, med et specifikt id, der blev givet til p5.js instansen.

### JavaScript

For at gøre disse elementer interaktive anvendte jeg JavaScript. Det jeg anvendte for at JavaScript havde adgang til HTML’en var JavaScripts document objekt. Document objektet i JavaScript symboliserer den HTML side JavaScripten indgår i, man kan ved dette objekt få adgang til alt i HTML’en, ved at bruge document objektets metoder. En af de metoder jeg anvendte mest hyppigt var document.getElementById, hvor man kalder metoden med en streng med Id’et på det ønskede HTML element. Derfra kan man så tilgå alle HTML elementets attributter, hvor dem jeg hyppigst tilgik fra HTML elementets value attribut. Jeg anvendte denne metode både til at hente data fra HTML’en, men også for at ændre på den. Denne metode blev anvendt 17 gange i mit program:



*Figur 18: Skærmbillede af søge værktøjet i VisualStudio Code, der viser hvor mange gange getElementById er anvendt.*

Ovenstående blev brugt til at få information fra HTML elementerne, men for at få siden til at reagere på klik på knapperne og lign. anvendte jeg HTML elementernes egne onclick eller onchange deklarationer, hvor jeg derfra kaldte en JavaScript funktion, der gjorde de ønskede ændringer. Derudover anvendte jeg også en JavaScript funktion kaldet setInterval til AutoStepthrough funktionens funktionalitet. setInterval funktionen virker ved at man giver den et interval (tid mellem hvert kald) og en funktion så sørger den for at kalde den funktion i den angivne frekvens.

### CSS

Til CSS delen anvendte jeg et bredt anvendt CSS-framework kaldt Bootstrap. Bootstrap er i bund og grund en lang CSS fil, med den har de dog lavet et let overskueligt system til at kunne lave hjemmesider, der virker godt på både mobil og computer. Jeg anvendte langt fra Bootstrap til dets fulde potentiale, men udover de basale formateringer anvendte jeg en af deres CSS-klasser til min tape, hvor jeg anvendte div klassen ”table-responsive”. Derudover lavede jeg nogle få udvidelser både i en separat CSS-fil kaldet styles.css, der ligger i data/style folderen sammen med den øvrige CSS, som indeholder følgende[[1]](#footnote-1):

Et billede, der indeholder skærmbillede, sort, skærm, rød

Automatisk genereret beskrivelse

*Figur 19: Det eneste CSS-kode jeg selv skrev.*

Derudover skrev jeg også en smule ”inline” CSS i HTML taggets style deklaration. Herunder ses hvordan jeg anvendte dette til mine knapper.

Et billede, der indeholder tekst, skilt, sort, grøn

Automatisk genereret beskrivelse

*Figur 20: Eksempel på CSS-kode direkte i HTML elementerne.*

# Program test

Under hele udviklingen sørgede jeg for at teste hvad jeg var i gang med. Jeg vil kort gennemgå hvilke og hvordan jeg testede forskellige dele af mit program. Til testen af fortolkeren testede jeg først hver af metoderne i klassen selv, derefter da fortolkeren var færdig sørgede jeg for at teste hver af funktionerne igennem og se om det forventede resultat kom ud. For alle undtagen ”[” og ”]” var dette relativt lige til, men for disse prøvede jeg flere af de mulige situationer. Da jeg havde testet om alle delene virkede, kørte jeg en række BF eksempelprogrammer for at se at de også virkede på større programmer. Ift. det grafiske startede jeg med at teste p5.js tegne funktioner direkte i koden. Derefter prøvede jeg at kalde samme funktioner fra en af de andre filer, hvilket afslørede en bug, der gjorde jeg skulle lave en smule om på min arkitektur. Da jeg havde testet tegnefunktionerne hver for sig, gjorde det det nemmere at teste stakken da jeg vidste fejlene associeret med de test hovedsageligt ville omhandle Stakken. Til Stakken sørgede jeg for at teste hver af opcodesne først gennem kode direkte og derefter gennem BF-fortolkeren. Den sidste del af programmet nemlig brugergrænsefladen testede jeg løbende ved bare at trykke på knapperne og se om det jeg forventede ville ske skete og ellers debugge.

# Konklusion

Da mit program lever op til alle mine krav og kommer igennem mine test, vil jeg sige at projektformuleringen er opfyldt. Det er den fordi mit program gør det muligt at lave BF programmer, der producere interaktiv grafik. Dog er der en lang række forbedringer/udvidelser jeg kunne forestille mig. Disse er følgende: understøtte alle p5.js tegnefunktioner, understøtte tekst i Stakken, mulighed for at definere tegnesekvenser (der kunne få sin egen opcode) og sidst udvide stakken til en mere Forth lignende stack, hvor at resultater ville gemmes på stakken, hvor de så enten kunne poppes af eller blive som argument. Dette ville nok også kræve at man lavede om på argument/opcode rækkefølgen, men ville gøre det langt mere udtryksfuldt. Dog ville mange nok ikke anvende dette da BF ikke er et særlig venligt sprog i det hele taget.

# Bilag

## Bilag 1 - index.html

<html>

<head>

<meta charset="utf-8"/>

<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=IBM+Plex+Mono"/>

<link rel="stylesheet" href="data/style/bootstrap.min.css">

<link rel="stylesheet" href="data/style/styles.css">

<script src="data/p5.js"></script>

<script src="bf.js"></script>

</head>

<body>

<script>

//Globale variabler

**let** mainInterval;

**let** intervalSpeed = **100**;

**let** program;

**let** interp;

**let** examplePrograms;

**let** doing\_step\_through = **false**;

**let** pause = **true**;

//Vi laver p5 instansen

**let** sketch = **function**(p) {

p.setup = **function**(){

p.createCanvas(**255**, **255**);

p.background(**0**);

p.fill(**255**,**0**,**0**);

}}

**let** p = **new** p5(sketch, window.document.getElementById('drawing\_window'));

//Dette er vores grafisk stack, den skal laves her for at have adgang til p5's funktioner

**class** Stack{

constructor(interpreter){

//En stack til at holde selve de skubbede opcodes og argumenter

**this**.stack = [];

//En instans af interpreteren den selv kører i for at kunne sætte værdier på tapen

**this**.interpreter = interpreter;

}

pushOPC(OPC){

//Det der bliver skubbet på stakken er en liste hvor det første element er opcoden og det andet er en liste af argumenter

**this**.stack.push([OPC, []]);

//Vi tjekker om vi kan køre opcoden allerede, da der er nogle opcodes, der ikke bruger nogle argumenter

**switch** (OPC){

//OPC: 8 -> keyIsPressed, hvilket bare er keyIsPressed værdien, der sættes på tapen

**case** **8**:

**this**.interpreter.tape[**this**.interpreter.tapeIndex] = p.keyIsPressed;

**this**.stack.pop();

**break**;

//OPC: 9 -> key, hvilket bare er key værdien lavet til et integer, der sættes på tapen

**case** **9**:

**this**.interpreter.tape[**this**.interpreter.tapeIndex] = parseInt(p.key);

**this**.stack.pop();

**break**;

//OPC: 10 -> keyCode, hvilket bare er keyCode værdien, der sættes på tapen

**case** **10**:

**this**.interpreter.tape[**this**.interpreter.tapeIndex] = p.keyCode;

**this**.stack.pop();

**break**;

//OPC: 11 -> mouseIsPressed, hvilket bare er mouseIsPressed værdien, der sættes på tapen

**case** **11**:

**this**.interpreter.tape[**this**.interpreter.tapeIndex] = p.mouseIsPressed;

**this**.stack.pop();

**break**;

//OPC: 12 -> mouseX, hvilket bare er mouseX værdien, der sættes på tapen

**case** **12**:

**this**.interpreter.tape[**this**.interpreter.tapeIndex] = p.mouseX;

**this**.stack.pop();

**break**;

//OPC: 13 -> mouseY, hvilket bare er mouseY værdien, der sættes på tapen

**case** **13**:

**this**.interpreter.tape[**this**.interpreter.tapeIndex] = p.mouseY;

**this**.stack.pop();

**break**;

}

}

pushARG(ARG){

//Når der skubbes et argument propper vi argumentet i den øverste opcode's argument liste

**let** opc = **this**.stack[**this**.stack.length-**1**][**0**];

**this**.stack[**this**.stack.length-**1**][**1**].push(ARG);

//Her har vi opcodens argument liste

**let** opc\_args = **this**.stack[**this**.stack.length-**1**][**1**];

//Her tjekker vi om vi kan køre nogle af opcodesne, det kan vi hvis de har det rette antal argumenter

**switch** (opc){

//OPC: 0 -> resizeCanvas(w, h)

**case** **0**:

**if**(opc\_args.length==**2**){

p.resizeCanvas(...opc\_args);

**this**.stack.pop();

**break**;

}

**else**{**break**}

//OPC: 1 -> fill(v)

**case** **1**:

**if**(opc\_args.length==**1**){

p.fill(...opc\_args);

**this**.stack.pop();

**break**;

}

**else**{**break**;}

//OPC: 2 -> fill(r, g, b)

**case** **2**:

**if**(opc\_args.length==**3**){

p.fill(...opc\_args);

**this**.stack.pop();

**break**;

}

**else**{**break**}

//OPC: 3 -> background(v)

**case** **3**:

**if**(opc\_args.length==**1**){

p.background(...opc\_args);

**this**.stack.pop();

**break**;

}

**else**{**break**}

//OPC: 4 -> background(r, g, b)

**case** **4**:

**if**(opc\_args.length==**3**){

p.background(...opc\_args);

**this**.stack.pop();

**break**;

}

**else**{**break**}

//OPC: 5 -> rect(x, y, w, h)

**case** **5**:

**if**(opc\_args.length==**4**){

p.rect(...opc\_args);

**this**.stack.pop();

**break**;

}

**else**{**break**}

//OPC: 6 -> circle(x, y, d)

**case** **6**:

**if**(opc\_args.length==**3**){

p.circle(...opc\_args);

**this**.stack.pop();

**break**;

}

**else**{**break**}

//OPC: 7 -> triangle(x1, y1, x2, y2, x3, y3)

**case** **7**:

**if**(opc\_args.length==**6**){

p.triangle(...opc\_args);

**this**.stack.pop();

**break**;

}

**else**{**break**}

}

}

}

//Man kan først få adgang til disse elementer efter at de er etableret i DOM'en

window.onload = **function** () {

//Her sætter vi nogle af GUI elementernes startværdier

examplePrograms = document.getElementById("examples");

program = examplePrograms.options[examplePrograms.selectedIndex].value;

document.getElementById("code").value = program;

//Vi initiere vores interpreter med stack

interp = **new** InterpreterWithStack(program);

changeSpeed();

};

//Denne kører koden der står i kode vinduet på "en gang" og propper resultatet i output paragraph

//Denne funktion kan ikke kaldes fra GUI'en.

**function** run(){

program = document.getElementById("code").value;

interp = **new** InterpreterWithStack(program);

interp.stack = **new** Stack(interp);

p.resizeCanvas(**255**,**255**);

p.background(**0**);

p.fill(**255**);

interp.eval();

document.getElementById("output").innerHTML = interp.output;

updateTape();

}

//Denne funktion opdatere tabellen i HTML'en, der repræsentere vores intepreters tape

**function** updateTape(){

//Hvis vores tape index er større en antallet af kollonner i tabllen

**if** (interp.tapeIndex >= **9**){

//Sætter vi værdierne fra fem til højre for tapeIndexet, da vores tapeIndex vil være i den miderste kolonne

**for** (**let** i = **0**; i < **10**; i++){

**let** cell = document.getElementById("t"+i.toString())

cell.innerHTML = interp.tape[interp.tapeIndex+i-**5**];

cell.style.backgroundColor = "#bbb4b4";

}

//Vi highlighter den miderste kollonne

document.getElementById("t5").style.background="red";

}

//Hvis vores tape index ikke er er større en antallet af kollonner i tabllen

**else**{

//Sætter vi bare de første værdier hele vejen igennem

**for** (**let** i = **0**; i < **10**; i++){

**let** cell = document.getElementById("t"+i.toString())

cell.innerHTML = interp.tape[i];

cell.style.backgroundColor = "#bbb4b4";

}

//Hvor vi bare highlighter den kollonne vores tapeIndex er i

document.getElementById("t"+interp.tapeIndex.toString()).style.backgroundColor="red";

}

}

//Denne funktion stopper vores AutoStepthrough

**function** pauseProgram(){

//Vi stopper hovedIntervallet og resetter værdien

clearInterval(mainInterval);

mainInterval = **null**;

}

//Denne funktion highlighter, det symbol, der køres lige nu

**function** highlight(){

**let** index = interp.programIndex;

//Det gør vi ved at redigere den indre HTML, hvor vi bare anvender <mark> tagget

innerHTML = program.substring(**0**,index-**1**) + "<mark style='font-size:25px'>" + program.substring(index-**1**,index) + "</mark>" + program.substring(index);

//Hvor vi så sætter den nye HTML ind på dets plads

document.getElementById("code\_text").innerHTML = innerHTML;

}

**function** stepThrough(){

//Er vi igang med at gå trinvist igennem

**if** (doing\_step\_through){

//Vi kalder selvfølgelig step

interp.step();

//Derefter opdatere vi nogle af GUI elementerne

document.getElementById("output").innerHTML = interp.output;

updateTape();

highlight();

//Vi tjekker om programmet er færdigt så det ikke bliver ved med at køre og så stopper vi det, hvis det var igang med autostepthrough

**if**(interp.done){

doing\_step\_through = **false**;

pauseProgram();

interp.done = **false**;

}

}

//Hvis ikke vi er igang med at gå trinvist igennem

**else**{

program = document.getElementById("code").value;

interp = **new** InterpreterWithStack(program);

interp.stack = **new** Stack(interp);

p.resizeCanvas(**255**,**255**);

p.background(**0**);

p.fill(**255**);

interp.step();

document.getElementById("output").innerHTML = interp.output;

updateTape();

doing\_step\_through = **true**;

document.getElementById("code\_text").innerHTML = program;

highlight();

}

}

**function** autoStepthrough(){

//Vi starter intervallet, hvis det ikke allerede er igang

**if** (mainInterval == **null**){

mainInterval = setInterval(stepThrough, intervalSpeed);

}

}

**function** changeProgram(){

//Hvis der vælges et andet eksempel program, så skifter vi koden i kode vinduet

program = examplePrograms.options[examplePrograms.selectedIndex].value;

document.getElementById("code").value = program;

doing\_step\_through = **false**;

}

**function** changeSpeed(){

//Vi får den ønskede intervalspeed fra slideren og sætter intervalSpeed variablen til denne værdi

intervalSpeed = document.getElementById("speed\_slider").value;

//Vi opdatere den viste hastighed

document.getElementById("speed\_display").innerHTML = "Time between each step: " + intervalSpeed.toString() + " ms";

}

</script>

<h1>Brainfuck with P5</h1>

<p>This was made as a school project. Find more information at <a href="https://github.com/simon220902/BrainFuckProcessing">the readme of my git.</a></p>

<h3>Tape</h3>

<!--Dette er vores tape-->

<div class="table-responsive">

<table class="table">

<thead style="background-color: #ffffff21">

<tr>

<th id="t0">0</th>

<th id="t1">0</th>

<th id="t2">0</th>

<th id="t3">0</th>

<th id="t4">0</th>

<th id="t5">0</th>

<th id="t6">0</th>

<th id="t7">0</th>

<th id="t8">0</th>

<th id="t9">0</th>

</tr>

</table>

</div>

<h3>Code</h3>

<textarea id="code" cols=100 rows=20></textarea>

<h3>Code being executed</h3>

<p id="code\_text"></p>

<!--Nedenstående var run knappen, der kørte hele programmet igennem i en køre, men dette virker ikke helt med nogle af de grafiske funktioner-->

<!--<button class="btn btn-primary" style="background-color:#bbb4b4;" onclick="run()">Run</button>-->

<button class="btn btn-primary" style="background-color:#1c6643;" onclick="stepThrough()">Step</button>

<button class="btn btn-primary" style="background-color:#1c6643;" onclick="autoStepthrough()">AutoStep</button>

<button class="btn btn-primary" style="background-color:#1c6643;" onclick="pauseProgram()">Pause</button>

<!--Eksempelprogrammerne, med deres værdi value feltet-->

<select id="examples" onchange="changeProgram()">

<option value="++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++." >Hello world</option>

<option selected="selected" value="Farve setup

- Vi sætter det første tal på tapen til 255

> Vi beholde det tal på tapen til 255

Nu laver vi så de basale koordinater vi regner med at skærmen er 255x255

Setup af koordinaterne

>>>

>->-

<<<<

Nu skal vi så rent faktisk lave værdierne

Vi skal gentage det 16 gange

++++++++++++++++

[>++++>++++++++>----<<<-]

Nu kan vi så begynde at tegne

>>>>>

Så laver vi baggrunden som skal være hvid (OPCODE 3)

+++:

<<<<<<<;>>>>>>>

Den første kommando vi gerne vil have er trekanten (altså OPCODE 7)

Men inden vi kan det skal vi første have farven til at være sort så altså fill (altså OPCODE 1)

--:<<<<<<;>>>>>>

Vi ændrer opcoden ++++++:

Nu skal vi så give den koordinaterne som er ( midt 1/4 fra toppen) (1/4 fra venstre 1/4 fra toppen) (1/4 fra højre 1/4 fra toppen)

( midt 1/4 fra toppen)

<<<;<;

(1/4 fra venstre midt)

;>;

(1/4 fra højre midt)

>;<;

Den anden kommando vi gerne vil have er firkanten til selve huset (altså OPCODE 5)

Men inden vi kan det skal vi første have farven til at være rød så altså fill men med rgb (altså OPCODE 2)

Vi starter med at gå hen til det rigtige sted >>> så får vi to -----: så går vi ned til starten<<<<<<<;>;;>>>>>>

Vi ændrer opcoden +++:

Nu skal vi så give den koordinatet og bredden og højden som er (1/4 fra venstre midt) Bredde halvdelen Højde 1/4 del

(1/4 fra venstre midt)

<<<<;>;

Bredde halvdelen af skærmen

;

Højde 1/4

<;

Den tredje kommando vi gerne vil have er firkanten til døren (altså OPCODE 5)

Men inden vi kan det skal vi første have farven til at være hvid så altså fill (altså OPCODE 1)

Vi starter med at gå hen til det rigtige sted >>>> så får vi et ----: så går vi ned til starten<<<<<<<; så går vi tilbage>>>>>>>vi ændrer opcoden til 5 ++++ :

Nu skal vi så give koordinatet og bredde og højde

vi skal ændre 1/4 del fra venstre til 3/8 dele fra venstre ved at plusse med 32

vi går til 1/4 del fra venstre <<<<

Så laver vi et loop der kører fire gange<++++

[>++++++++<-]sådan så har vi det>;

så skal vi også bare have midten>;

Nu har vi koordinaterne

Nu skal vi så give bredden der skal være 1/8 af skærmen altså 32

<<<++++

[>++++++++<-]

>;

Nu skal vi så bruge 64

<++++

[>++++++++<-]

>;">Hus</option>

<option value="ADVARSEL! Lad vær med at køre dette med run knappen kør det kun med AutoStepThrough ellers crasher det!!!

Okay så hvad er ideen at lave en cirkel der hvor musen er (cirkel OPCODE I mousex OPCODE I mousey OPCODE I cirkel r) Vi laver cirklens opcode ++++++ Vi laver cirklens radius >>+++++++ [>++++++++++<-] <<

[

vi starter med at skubbe tegnCirkel :

Nu skal vi så få musens X koordinat så vi laver den opcode (altså 12) > [-] ++++ ++++ ++++ :

Så erstattes værdien og vi skubber den på som argument ;

Nu skal vi så få musens Y koordinat så vi laver den opcode (altså 13) > [-] ++++ ++++ ++++ + :

Så erstattes værdien og vi skubber den på som argument ;

nu mangler vi bare radius > ;

og så går vi tilbage til tegn cirkels opcode og gør det samme igen <<<

]">Cirkler ved mus</option>

<option value="26 +++++ +++++ +++++ +++++ +++++ +>

A +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++

<

[>.+<-]">Alphabet</option>

</select>

<h4>Speed of AutoStep</h4>

<input type="range" min="0" max="500" step="1" value="100" class="slider" id="speed\_slider" onchange="changeSpeed()">

<p id="speed\_display"></p>

<h3>Output</h3>

<p id="output"></p>

<h3>Graphical output</h3>

<!--Vores p5 tegne vindues-->

<div id='drawing\_window'></div>

</body>

</html>

## Bilag 2 - bf.js

**class** Interpreter{

constructor(program){

//Vores tape består af 30000 8 bit unsigned ints

**this**.tape = **new** Uint8Array(**30000**);

**this**.tapeIndex = **0**;

//Vores program er bare en lang streng, der også indeholder symboler som intet gør

**this**.program = program;

**this**.programIndex = **0**;

//Loopstart bruges til at gemme indekset i programmet hvor løkker starter

**this**.loopStart = [];

//Outputtet fra programmet, der kommer fra "." symbolet gemmes i denne streng

**this**.output = "";

//Om den er i gang med gå trinvist igennem programmet

**this**.is\_stepping = **false**;

//Om programmet er færdig med at køre

**this**.done = **false**;

}

reset(){

//Vi resetter de nødvendige variablers værdier så programmet kan køre igen

**this**.tape = **new** Uint8Array(**30000**);

**this**.tapeIndex = **0**;

**this**.programIndex = **0**;

**this**.loopStart = [];

**this**.output = "";

**this**.is\_stepping = **false**;

}

eval(){

//Eval funktionen kører hele programmet.

//Hvis interpreteren er igang med en trinvis gennemgang resettes værdierne så den er klar til at køre programmet hele vejen igennem.

**if** (**this**.is\_stepping){

**this**.reset();

}

//For hver karakter i strengen, this.programIndex anvendes da det er denne pointer vi manipulere ift. håndteringen af løkker

**for**(;**this**.programIndex < **this**.program.length; **this**.programIndex++){

**switch**(**this**.program[**this**.programIndex]) {

//Et case for hver af de BF symboler

**case** "<":

**this**.left();

**break**;

**case** ">":

**this**.right();

**break**;

**case** "+":

**this**.plus();

**break**;

**case** "-":

**this**.minus();

**break**;

**case** ".":

**this**.printChar();

**break**;

**case** ",":

**this**.getChar();

**break**;

**case** "[":

**this**.loopOpening();

**break**;

**case** "]":

**this**.loopClosing();

**break**;

**default**:

}

}

}

step(){

//Step funktionen kører kun et enkelt trin, hver gang den bliver kaldt, altså kun et symbol.

//Hvis programmet ikke er igang med en trinvis gennemgang, resetter vi interpreteren og sætter den til at være igang

**if** (!**this**.is\_stepping){

**this**.reset();

**this**.is\_stepping=**true**;

}

//Så længe der er program tilovers

**else** **if**(**this**.programIndex< **this**.program.length){

//Har vi samme switch case for hver af BF symbolerne

**switch**(**this**.program[**this**.programIndex]) {

**case** "<":

**this**.left();

**break**;

**case** ">":

**this**.right();

**break**;

**case** "+":

**this**.plus();

**break**;

**case** "-":

**this**.minus();

**break**;

**case** ".":

**this**.printChar();

**break**;

**case** ",":

**this**.getChar();

**break**;

**case** "[":

**this**.loopOpening();

**break**;

**case** "]":

**this**.loopClosing();

**break**;

**default**:

}

//Vi tager det næste symbol

**this**.programIndex += **1**;

}

//Hvis der ikke er mere program tilovers er programmet færdigt, vi sætter værdien done til true og sætter til ikke at være igang med en trinvisgennemgang

**else**{

**this**.done = **true**;

**this**.is\_stepping = **false**;

}

}

//Herunder kommer funktionerne tilknyttet til hver af BF symboler

//Tilsvarende BF symbol: <

left(){

**this**.tapeIndex--;

}

//Tilsvarende BF symbol: >

right(){

**this**.tapeIndex++;

}

//Tilsvarende BF symbol: +

plus(){

**this**.tape[**this**.tapeIndex]++;

}

//Tilsvarende BF symbol: -

minus(){

**this**.tape[**this**.tapeIndex]--;

}

//Tilsvarende BF symbol: .

printChar(){

//Vi tilføjer værdien på tapen som et ASCII symbol til output strengen

**this**.output += String.fromCharCode(**this**.tape[**this**.tapeIndex]);

}

//Tilsvarende BF symbol: ,

getChar(){

//Vi får et input via en prompt

**let** **char** = prompt('Which character do you want to insert at '+**this**.tapeIndex, '');

//Vi sørger for at inputtet kun er en karakter lang og bliver ved med at spørge til vi får et input, der kun er en karakter lang

**while** (**char**.length !== **1**){

**char** = prompt('Please insert a single character only '+**this**.tapeIndex, '');

}

//Når inputtet er en karakter langt, sætter vi den tilsvarende ASCII værdi ind på den nuværende plads på tapen

**this**.tape[**this**.tapeIndex] = **char**.charCodeAt();

}

//Tilsvarende BF symbol: [

loopOpening(){

//Hvis værdien på den nuværende plads på tapen ikke er nul og loopet skal køre

**if** (**this**.tape[**this**.tapeIndex] !== **0**){

//Tilføjer vi programindekset til loopStart listen, der virker som en stack, hvor det nyeste loop åbning kommer til at ligge øverst

**this**.loopStart.push(**this**.programIndex);

}**else**{

//Hvis værdien på den nuværende plads på tapen er nul, skal loopet ikke køre, så vi finder, der hvor loopet stopper og sætter det til vores program index

**this**.programIndex = **this**.loopEnd(**this**.programIndex)+**1**;

}

}

//Tilsvarende BF symbol: ]

loopClosing(){

//Hvis værdien på den nuværende plads på tapen er nul og loopet er færdigt

**if** (**this**.tape[**this**.tapeIndex] === **0**){

//Tager vi loopets start index af loopStart listen

**this**.loopStart.pop();

}**else**{

//Ellers skal vi rykke vores programIndex tilbage til loopets start

**this**.programIndex = **this**.loopStart[**this**.loopStart.length-**1**];

}

}

//Hjælpefunktion til loopOpening

loopEnd(index){

//Denne hjælpefunktion finder et loops matchende slutning, det betyder altså også at den skal kunne håndtere loops inden i det første loop

//For alle de resterende karaktere i program strengen, vi slutter loopet hvis det ønskede resultat findes

**for** (**let** iadd=**1**;iadd+index<**this**.program.length;iadd++){

//Hvis der er en loop slutning så returnere vi det nye index

**if** (**this**.program[index+iadd]=="]"){

**return** index+iadd;

}//Hvis et nyt loop åbnes kalder vi funktionen igen, hvor vi finder slutningen på det nye åbnede loop

**else** **if**(**this**.program[index+iadd]=="["){

//Da loopEnd funktionen returnere index+iadd og vi kun er interesseret i iadd værdien trækker vi index fra.

iadd = **this**.loopEnd(index+iadd)-index;

}

}

}

}

**class** InterpreterWithStack **extends** Interpreter{

constructor(program){

//Da denne klasse er nedarvet af Interpreteren kalder vi interpreteren constructor med programmet

**super**(program);

//Den grafiske stack er det der er forskellen på denne underklasse så vi laver en instans af Stack klassen

//Vi giver stakken objektet selv så den kan proppe ting på tapen

**this**.stack = **new** Stack(**this**);

}

//Det samme som i Interpreter bare med to nye symboler ":" og ";"

eval(){

**for**(;**this**.programIndex < **this**.program.length; **this**.programIndex++){

**switch**(**this**.program[**this**.programIndex]) {

**case** "<":

**super**.left();

**break**;

**case** ">":

**super**.right();

**break**;

**case** "+":

**super**.plus();

**break**;

**case** "-":

**super**.minus();

**break**;

**case** ".":

**super**.printChar();

**break**;

**case** ",":

**super**.getChar();

**break**;

**case** "[":

**super**.loopOpening();

**break**;

**case** "]":

**super**.loopClosing();

**break**;

**case** ":":

**this**.pushOPC(**this**.tape[**this**.tapeIndex]);

**break**;

**case** ";":

**this**.pushARG(**this**.tape[**this**.tapeIndex]);

**break**;

}

}

}

//Det samme som i Interpreter bare med to nye symboler ":" og ";"

step(){

**if** (!**this**.is\_stepping){

**this**.reset();

**this**.is\_stepping=**true**;

}

**else** **if**(**this**.programIndex < **this**.program.length){

**switch**(**this**.program[**this**.programIndex]) {

**case** "<":

**super**.left();

**break**;

**case** ">":

**super**.right();

**break**;

**case** "+":

**super**.plus();

**break**;

**case** "-":

**super**.minus();

**break**;

**case** ".":

**super**.printChar();

**break**;

**case** ",":

**super**.getChar();

**break**;

**case** "[":

**super**.loopOpening();

**break**;

**case** "]":

**super**.loopClosing();

**break**;

**case** ":":

**this**.pushOPC();

**break**;

**case** ";":

**this**.pushARG();

**break**;

}

**this**.programIndex += **1**;

}

**else**{

**this**.done = **true**;

**this**.is\_stepping = **false**;

}

}

//Tilsvarende vores grafiske BF symbol: :

pushOPC(){

//Vi bruger stakkens metode hvor vi skubber værdien på den nuværende plads i tapen

**this**.stack.pushOPC(**this**.tape[**this**.tapeIndex]);

}

//Tilsvarende vores grafiske BF symbol: ;

pushARG(){

//Vi bruger stakkens metode hvor vi skubber værdien på den nuværende plads i tapen

**this**.stack.pushARG(**this**.tape[**this**.tapeIndex]);

}

}

## Bilag 3 - Flowchart over Step metode fra InterpreterWithStack

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

## Bilag 4 - Prædefinerede processer til flowchart over Step metode fra InterpreterWithStack

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

1. Her er et link til data/style/ i min git: <https://github.com/Simon220902/BrainFuckProcessing/tree/master/data/style> [↑](#footnote-ref-1)