

MIKKEL AABO SIMONSEN
WEI YANG
JOHANNES THOMASSEN

TETRIS I JAVASCRIPT

Abstract

Denne undersøgelse fokuserer på Tetris som et ikonisk retrospil og dets vedvarende popularitet. Vi undersøger spillets spændende funktionaliteter og algoritmer samt dets indflydelse på spilverdenen. Gennem en kombination af analyse og kreativ udvikling udforsker vi Tetris' betydning og potentiale for nutidige applikationer og spilplatforme.

1. Introduction

Tetris er fortsat en af de mest populære retrospil til dato. Derfor har vi besluttet os for at udvikle Tetris, da vi finder dets funktionaliteter og algoritmer særligt interessant.

I Tetris bliver der brugt en bred vifte af forskellige datastrukturer.
I vores projekt har vi herunder benyttet os af:

- Queue
- Grid
- Stack

Derudover har vi skulle løse en masse problemer, som har ledt til, at vi har måtte udarbejde en del algoritmer.

Herunder har vi udarbejdet løsninger til bl.a.

- Collision Check
- Rotation på 2-dimensionelt grid
- Line-clearing

Produktet samt kildekoden kan ses på github

<https://github.com/DizzyMoon/Tetris>

2. Datastrukturer

2.1. Queue

Siden JavaScript ikke understøtter queue datatypen, har vi været nødt til at implementere vores egen i form af en klasse *Queue*.

I denne klasse har vi implementeret standard Queue funktioner, til at læse og manipulerer køen.

De implementerede funktioner er som følgende:

- *enqueue(item)* - Indsætter *item* bagerst i køen.
- *dequeue()* - Fjerner det sidste element i køen.
- *peek()* - Returnerer det foreste element i Queue indstansen.
- *printQueue()* - Returnerer listen af items i køen.
- *length()* - Returnerer længden af vores queue, altså indexet af det foreste element minus indexet af det sidste element.

```
1 class Queue {
2     constructor() {
3         this.items = {};
4         this.frontIndex = 0;
5         this.backIndex = 0;
6     }
7
8     enqueue(item) {
9         this.items[this.backIndex] = item;
10    this.backIndex++;
11    return item + " inserted";
12 }
13
14 dequeue() {
15    const item = this.items[this.frontIndex];
16    delete this.items[this.frontIndex];
17    this.frontIndex++;
18    return item;
19 }
20
21 peek() {
22    return this.items[this.frontIndex];
23 }
24 get printQueue() {
25    return this.items;
26 }
27
28 get length() {
29    return this.backIndex - this.frontIndex;
30 }
31 }
32
33 export default Queue;
```

2.2. PieceQueue

Til opbevaring af de næste aktive Tetris figurer, bruger vi en simpel queue datastruktur.

På den måde kan vi tjekke hvad den næste figur er, ved at bruge *peek()* funktionen, defineret i Stack klassen

Denne Stack klasse bliver implementeret i en PieceQueue klasse, hvilket gør det muligt at tilføje funktionalitet unikt til den instans af Stack.

Vi bruger f.eks. en funktion: *drawQueue()*, som tegner det forreste objekt i PieceStack.pieces med html elementer i index.html.

pieces ← *newStack*

DrawQueue() er også ansvarlig for, at sørge for at *pieces* altid har piece elementer. Dette gøres med et simpelt for-loop.

```
1  drawQueue() {
2    const queueContainer = document.getElementById("queue-container");
3    queueContainer.innerHTML = "";
4    const pieceQueue = document.createElement("div");
5    pieceQueue.classList.add("piece-queue");
6    queueContainer.appendChild(pieceQueue);
7
8    for (let i = 0; i < 10; i++) {
9      let piece = new Piece();
10     const type = piece.getRandomType();
11     piece.type = type;
12     this.pieces.enqueue(piece);
13   }
14
15   const frontPiece = this.pieces.peek();
16   const img = document.createElement("img");
17   img.classList.add("queue-piece");
18
19   switch (frontPiece.type) {
20     case "square":
21       img.src = "./images/pieces/Square piece.png";
22       break;
23     case "line":
24       img.src = "./images/pieces/Line piece.png";
25       break;
26     case "z":
27       img.src = "./images/pieces/Z piece.png";
28       break;
29     case "z-reverse":
30       img.src = "./images/pieces/Z piece inverted.png";
31       break;
32     case "l":
33       img.src = "./images/pieces/L piece.png";
34       break;
35     case "l-reverse":
36       img.src = "./images/pieces/L piece inverted.png";
37       break;
38     case "t":
39       img.src = "./images/pieces/T piece.png";
40       break;
41   }
42
43   pieceQueue.appendChild(img);
44 }
```

2.3. Stack

Af samme årsager som med Queue datastrukturen, har vi været nødt til at implementere vores egen Stack klasse, med de følgende funktioner:

- *add(element)* - Tilføjer element Forrest i stakken.
- *remove()* - Fjerner det forreste element fra stakken.
- *peek()* - Returnerer det forreste element i stakken.
- *isEmpty()* - Returnerer true, hvis stakken er tom og ellers false.
- *size()* - Returnerer mængden af elementer i stakken.
- *clear()* - Tømmer stakken.

```
1 class Stack {
2     constructor() {
3         this.items = [];
4     }
5
6     add(element) {
7         return this.items.push(element);
8     }
9
10    remove() {
11        if (this.items.length > 0) {
12            return this.items.pop();
13        }
14    }
15
16    peek() {
17        return this.items[this.items.length - 1];
18    }
19
20    isEmpty() {
21        return this.items.length === 0;
22    }
23
24    size() {
25        return this.items.length;
26    }
27
28    clear() {
29        this.items = [];
30    }
31}
32
33 export default Stack;
```

2.4. PieceStack

PieceStack minder meget om PieceQueue. Dens eksistens er næsten udelukkende pga. dens *drawStack()* funktion, som tegner det forreste element i stakken på brugerfladen, med html elementer.

I denne funktion tjekker vi først om der findes pieces i stakken, hvis der gør, gemmer vi det sidste element fra stakken i variablen frontPiece, som anvendes i en switch case. Ud fra resultatet af switch casen, indsætter vi den korrekte figur i stack html elementet i form af et billede af figuren

```
1  drawStack() {
2    const stackContainer = document.getElementById("stack-container");
3    stackContainer.innerHTML = "";
4    const pieceStack = document.createElement("div");
5    pieceStack.classList.add("piece-stack");
6    stackContainer.appendChild(pieceStack);
7
8    if (this.pieces.peek()) {
9      const frontPiece = this.pieces.peek();
10     const img = document.createElement("img");
11     img.classList.add("stack-piece");
12
13     switch (frontPiece) {
14       case "square":
15         img.src = "./images/pieces/Square piece.png";
16         break;
17       case "line":
18         img.src = "./images/pieces/Line piece.png";
19         break;
20       case "z":
21         img.src = "./images/pieces/Z piece.png";
22         break;
23       case "z-reverse":
24         img.src = "./images/pieces/Z piece inverted.png";
25         break;
26       case "l":
27         img.src = "./images/pieces/L piece.png";
28         break;
29       case "l-reverse":
30         img.src = "./images/pieces/L piece inverted.png";
31         break;
32       case "t":
33         img.src = "./images/pieces/T piece.png";
34         break;
35     }
36     pieceStack.appendChild(img);
37   }
38 }
```

3. Rotations algoritme

3.1. Teori

Det helt store problem i at rekreere Tetris vi stødte ind i, var selvfølgelig:

"Hvordan roterer man en Tetris figur i en 2d matrise.

Der findes 7 forskellige Tetris figurer, som skal kunne roteres 90 grader med uret, hver gang spilleren trykker på roter knappen. Alle disse forskellige figurer har et center punkt, som vi skal roterer rundt om

Vi ved at den generelle formel for at rotere et punkt rundt om et centrum er:

$$x' = x \cdot \cos(\theta) - y \cdot \sin(\theta)$$

$$y' = x \sin(\theta) - y \cdot \cos(\theta)$$

Da vores figurer ikke har centrum i koordinatet (0, 0), er vi nødt til at definerer et nyt centrum

$$x_{new} = x - a$$

$$y_{new} = y - b$$

$$x \leftarrow x_{new}$$

$$y \leftarrow y_{new}$$

Nu når vi har en måde at beregne de nye koordinater for en roteret figur, kan vi gøre dette for hver *"blok"* i Tetris figuren.

Vi kan altså definere følgende funktion:

```
function ROTATE(points, origin, θ)
    rotatedPoints ← {};
    for x0 ← 0 to points.size() − 1 do
        x0 ← points[i].x − origin.x;
        y0 ← points[i].y − origin.y;
        x' ← x0 · cos(θ) − y0 · sin(θ);
        y' ← x0 · sin(θ) + y0 · cos(θ);
        x' ← x' + origin.x;
        y' ← y' + origin.y;
        rotatedPoints.add(x', y');
    end for
    return rotatedPoints
end function
```

3.2. Implementation

Nu når vi har pseudokoden på plads, kan vi implementere det i Javascript. I vores implementation, valgte vi at beregne centrum inde i funktionen ved at finde det de gennemsnitlige koordinater af alle blokke i en figur.

På den måde behøver rotate funktionen ikke at modtage origin som parameter.

```
1  function rotate(pieceList, degrees) {
2    let radians = (degrees * Math.PI) / 180; // Convert degrees to radians
3    let rotatedPieces = [];
4
5    let origin = pieceList.reduce(
6      (acc, piece) => [acc[0] + piece.position[0], acc[1] + piece.position
7      [1]],
8      [0, 0]
9    );
10   origin[0] /= pieceList.length;
11   origin[1] /= pieceList.length;
12
13   for (let i = 0; i < pieceList.length; i++) {
14     let x0 = pieceList[i].position[0] - origin[0];
15     let y0 = pieceList[i].position[1] - origin[1];
16
17     let xPrime = Math.round(
18       x0 * Math.cos(radians) - y0 * Math.sin(radians) + origin[0]
19     );
20     let yPrime = Math.round(
21       x0 * Math.sin(radians) + y0 * Math.cos(radians) + origin[1]
22     );
23
24     const rotatedPiece = {
25       color: pieceList[i].color,
26       position: [xPrime, yPrime],
27       current: pieceList[i].current,
28       type: pieceList[i].type,
29     };
30     rotatedPieces.push(rotatedPiece);
31   }
32   return rotatedPieces;
33 }
```

Det er ikke nok bare at rotere figuren, vi er også nødt til at tjekke om den beregnede rotation overhovedet kan lade sig gøre.

Der kan let opstå situationer, hvor de felter vi roterer til, enten ligger udenfor grid elementet, eller at de allerede er optaget af andre figurer.

Disse tilfælde er vi nødt til at håndtere, så vi ikke løber ind i eventuelle bugs og crashes:

```
1  canRotate(rotatedPieces) {
2    let canRotate = true;
3
4    rotatedPieces.forEach((piece) => {
5      if (piece.position[0] <= 0 || piece.position[0] >= this.columns) {
6        canRotate = false;
7      }
8      if (piece.position[1] <= 0 || piece.position[1] >= this.rows) {
9        canRotate = false;
10     }
11
12     if (this.isOccupied(piece.position[0], piece.position[1])) {
13       canRotate = false;
14     }
15   });
16
17   return canRotate;
18 }
```

I denne funktion returnerer vi true, hvis en rotation for de pågældende roterede blokke, vi har fået ind som parameter kan lade sig gøre, og omvendt false, hvis det ikke kan lade sig gøre.

4. Line-clear algoritme

Man kan jo ikke lave et Tetris spil uden at kunne fjerne horizontale linjer, der er fyldt helt ud. Ellers bliver det ihvertfald meget svært at vinde. Det vil sige at vi er nødt til at skrive en algoritme, som kan iterere igennem hele grid elementet, og tjekke om nogle af rækkerne er fyldte:

```
function CLEARLINES

    for i ← 0 to grid.length – 1 do
        row ← grid[i]
        lineCompleted ← true
        for j ← 0 to row.length – 1 do
            if row[j] = empty then
                lineCompleted ← false
            end if
        end for
        if lineCompleted then
            for j ← 0 to row.length – 1 do
                grid[i][j] ← empty
            end for
            for k ← i to 0 do
                grid[k] ← grid[k – 1]
            end for
            InitializeGridRowWithEmpty
        end if
    end for
```

Denne funktion tjekker hvert enkelte række og indsætter *empty* på alle felter i rækken, hvis der ikke findes et felt, som allerede er *empty*.

I vores implementation af denne algoritme gemmer vi også mængden af linjer, som er blevet slettet, så det er muligt at tælle point efter algoritmen er blevet kørt:

```
1  clearLines() {
2    let number_of_lines = 0;
3
4    for (let i = 0; i < this.grid.length; i++) {
5      const row = this.grid[i];
6      let lineCompleted = true;
7      for (let j = 0; j < row.length; j++) {
8        if (row[j] === null) {
9          lineCompleted = false;
10         break;
11       }
12     }
13     if (lineCompleted) {
14       number_of_lines++;
15       for (let j = 0; j < row.length; j++) {
16         this.grid[i][j] = null;
17       }
18       for (let k = i; k > 0; k--) {
19         this.grid[k] = this.grid[k - 1];
20       }
21       this.grid[0] = Array(this.columns).fill(null);
22       this.drawGrid("grid-container");
23       break;
24     }
25   }
26
27   this.totalClearedLines += number_of_lines; // Accumulate the count of
28   cleared lines
}
```

5. Konklusion

Efter at have arbejdet med Tetris i et par uger har vi fået en grundig gennemgang af udarbejdning og implementering af forskellige algoritmer i praksis, samt identificere intelligent brug af relevante datastrukturer.

Det er både lykkedes os, at identificere et problem i form af, hvordan vi håndterer:

- Kollision
- Line-clear
- Rotation

og derefter finde en fornuftig løsning på problemet i form af algoritmer enten udarbejdet fra matematisk teori, eller anden vis.

Samtidig har det været et super sjovt projekt at arbejde på, da det altid er spændende at udvikle et produkt fra start til slut.