Les petits pavés

Guillaume Pelletier-Auger

21 mars 2017

Résumé

Explorations diverses des pavages de Truchet.

1 Introduction

Ceci est l'introduction à mon projet sur les pavages de Truchet.

2 Réflexions diverses

Je dois penser à mes systèmes de Truchet comme à des espaces mathématiques dans lesquels je peux me déplacer, un peu comme l'ensemble de Mandelbrot est un espace mathématique. Je peux créer des *ensembles de Truchet* que mon système pourra visualiser à n'importe quelles valeurs.

Idéalement, il me faudrait pouvoir définir mes ensembles de Truchet ainsi : Un groupe de 4 permutations : A, C, B, A, miroité horizontalement avec A, B, C, A, et miroité verticalement avec X, X, X, X.

Essentiellement, je dois pouvoir définir un bloc, de telle largeur et telle hauteur, et ensuite, définir ses répétitions et, optionnellement, ses symétries.

```
var block = {
    lines  : [ABBACDAB, CDDBDADD],
    horizontalSymmetry : true,
    verticalSymmetry : false,
    diagonalSymmetry : false
};
```

La combinaison de la symétrie horizontale et de la symétrie verticale crée automatiquement une symétrie diagonale.

3 Transitions animées entre les positions des tuiles

Il me faut absolument créer des transitions animées entre les diverses positions que les tuiles de Truchet peuvent prendre. Il pourrait y avoir des transitions dans le sens des aiguilles d'une montre et des transitions dans le sens inverse. L'usage de ces 2 sens pourrait créer des effets variés, et même être combinés (mais je ne sais pas exatement comment, pour l'instant).

J'ai même l'impression que tout est peut-être là. Tout le charme du film pourrait être dans ces transitions-là. Parce que des animations de pavages de Truchet sans transitions, comme je l'ai expérimenté hier soir (20 mars 2017), ça fait très froid. Ça fait peu musical, également.

4 Les différents morceaux

4.1 Une feuille d'instructions

Il me faut une feuille d'instructions, qui est essentiellement un espace mathématique. Une feuille d'instruction contient : une largeur et une hauteur en nombres entiers, une booléenne *horizontalSymmetry* et une booléenne *verticalSymmetry*. Et finalement, une feuille d'instruction contient soit des données statiques qui remplissent sa largeur et sa hauteur, soit un lien vers une autre feuille d'instruction, avec une valeur *offset* qui sera un vecteur en nombre entier.

Donc, ça donne ça

```
//First type of block, with static data.
var blockOne = {
    type  : "static",
    size  : {x : 8, y : 4},
    data  : ["ABBACDAB", "CDDBDADD", "CDABCDAB", "DADDDADD"],
    horizontalSymmetry : true,
    verticalSymmetry : false
};

//Second type of block, with a reference to another block.
var blockTwo = {
    type  : "dynamic",
    size  : {x : 8, y : 2},
    data  : {
        ref : blockOne,
        offset : {x : 50, y : 50}
    },
    horizontalSymmetry : true,
    verticalSymmetry : true
};
```

Et si je créais un bloc ainsi ? Est-ce qu'un bloc devrait avoir son propre array *fullSpace* ? Dans tous les cas, lorsque je me déplace à l'intérieur de l'espace mathématique d'un bloc, je ne devrais pas avoir à recalculer chaque fois chacune des cellules de cet espace mathématique. Il faut que ce grand array *fullSpace* existe quelque part. Et comme il est, conceptuellement, une propriété du bloc, il devrait peut-être devenir factuellement une propriété du bloc.

Le seul problème que j'envisage avec ce système, c'est le cas des blocs dynamiques. Les blocs dynamiques créent forcément des valeurs *fullSpace* différentes. Il faudrait peut-être simplement que les blocs dynamiques aient une méthode *reCalculateFullSpace*. Étrangement, cette fonction se retrouverait à scruter dans la valeur *fullSpace* d'un autre bloc. Ainsi, animer un bloc dynamique, ça voudrait dire changer sa valeur *offset* dans le temps, ce qui ferait que ce bloc scruterait un autre bloc à une position spatiale changeante.

```
var Block = function(block) {
    this.type = block.type;
    this.size = block.size;
    this.data = block.data;
    this.horizontalSymmetry = block.horizontalSymmetry;
    this.verticalSymmetry = block.verticalSymmetry;
    this.fullSpace = this.makeLargeSpace(block)
};

Block.prototype.makeLargeSpace = function(block) {
    var largeArray = [];
    //Make copies to take care of symmetries,
    //to end up with blocks that can be copied simply without symmetry.

    //And then, make a large amount of copies.
    //Return the large array.
};

var blockOne = new Block({
    type  : "dynamic",
    size  : {x : 8, y : 2},
    data  : {
        ref : blockOne,
        offset : {x : 50, y : 50}
    },
    horizontalSymmetry : true,
    verticalSymmetry : true,
    maxSize  : {x : 500, y :700}
});
```

4.2 Une fonction chercheuse

Il me faut une fonction chercheuse, qui posera ses questions aux feuilles d'instructions.

```
// Voici comment je dois pouvoir appeler la fonction chercheuse.
sheetSearcher({
    startX : 50,
```

```
    startY : 50,
    ref : blockOne
});

//Mais comment puis-la définir ?
function sheetSearcher(instructions) {
    var truchetValue;
    if (instructions.startX == 0 && instructions.starty == 0) {
        truchetValue = instructions.ref.data[0][0];
    }
    return truchetValue;
}
```

La fonction chercheuse reçoit la question suivante : À l'endroit x = 50, y = 50, quelle est la valeur Truchet (A, B, C ou D) de *blockOne* ? Elle commence à répondre à cette question en posant la question à *blockOne* : à la valeur x = 0, y = 0, quelle est ta valeur Truchet ? Et ensuite, si j'augmente le x de 1, quelle est ta valeur Truchet ? La fonction chercheuse commence à trouver la valeur x = 50, y = 0, et ensuite, elle se déplace sur l'axe Y, toujours en posant une question à *blockOne* pour chaque déplacement unitaire. Les booléennes de *blockOne* sont évidemment déterminantes du comportement de la fonction chercheuse. Elle se sert de ces booléennes pour se déplacer à l'intérieur de l'espace mathématique créé par la feuille d'instructions.

Une bien meilleure idée pour l'algorithme de la fonction chercheuse : Elle va se servir des instructions de la feuille d'instruction pour créer un énorme pavage (en copiant les données et créant un énorme array), et ensuite elle se servira de ce pavage pour créer le pavage final. Elle ne génère ainsi les données qu'une seule fois.

4.3 Une fonction tilingFiller

Il me faut une fonction qui remplisse un pavage *final*. Cette fonction utilise la fonction chercheuse.

```
// Voici comment je dois pouvoir appeler la fonction remplissante.
var myTiling = tilingFiller({
    size : {width : 32, height : 18},
    ref : blockOne,
    offset : {x : 10, y : 10}
});

// Et la définition de cette fonction.
function tilingFiller(instructions) {
    var tiling = [];
    for (var x = 0; x < instruction.size.width; x += 1) {
        for (var y = 0; y < instruction.size.height; y += 1) {
            var truchetValue = sheetSeeker({
                startX : instructions.offset.x,
                startY : instructions.offset.y,
                ref : instructions.ref
            });
            //Important de réviser ceci avant l'usage  :
            tiling[x + y * instructions.size.width] = truchetValue;
        }
```

```
    }
    return tiling;
}

//Version diaboliquement plus simple de tilingFiller()  :

var myTiling = tilingFiller({
    inputBlock : blockOne,
    offset : {x : 70, y : 70},
    outputSize  : {width : 32, height :18}
});

function tilingFiller(instructions) {
    var block = instructions.inputBlock;
    var offset = instructions.offset;
    var size = instructions.outputSize;
    var tiling = [];
    for (var x = 0; x < size.width; x += 1) {
        for (var y = 0; y < size.height; y += 1) {
            tiling[x + y * size.width] = block.fullSpace[(offset.x + x) + ((offset.y + y) * size.width)];
        }
    }
    return tiling;
}
```

4.4 Des zones alpha animables pour les pavages

Il me faut une façon de définir essentiellement la zone alpha et la position d'un pavage, qui pourront être animées par le système d'animation.