La nuit géométrique

Guillaume Pelletier Auger

1^{er} mai 2016

Résumé

Un film d'animation programmé en JavaScript avec l'aide de la bibliothèque p5.js.

1 Introduction

Le film doit n'avoir qu'une seule scène de 12 minutes. Pas de coupes, juste des transitions douces. Donc il me faut une fonction qui gère le temps qui passe, et qui, selon la durée écoulée (la valeur frameCount), assigne les points du graphe à des oscillateurs différents. Je vais appeler mes fonctions, ou mes scènes, ainsi : des oscillateurs.

Un oscillateur peut n'être qu'une simple fonction paramétrique, mais il doit contrôler lui-même la valeur m. Un oscillateur doit pouvoir prendre les données qui sont présentement dans le graphe g et en faire une interpolation linéaire vers sa propre *forme oscillante*. Un oscillateur doit pouvoir être multiple, c'est-à-dire varier entre 2 équations paramétriques par une interpolation linéaire qu'il contrôle lui-même. Un oscillateur doit pouvoir contrôler l'output vers les fonctions de coloration et effets spéciaux. Un oscillateur doit pouvoir créer des faux *mouvements de caméra*, c'est-à-dire déplacer les sommets du graphe dans n'importe quelle direction à n'importe quelle vitesse.

2 Structure

2.1 Le prototype d'oscillateur

Tous les oscillateurs sont des objets qui héritent du même prototype, l'objet *Oscillator*. Le prototype a deux méthodes : *update* et *mix*. *Update* met à jour les valeurs x et y de l'objet en utilisant la valeur globale *drawCount* pour faire rouler l'équation paramétrique qui est stockée dans la valeur *eq* de l'objet. *Mix* fait la même chose, mais en plus, cette méthode fait une interpolation linéaire entre les valeurs x et y de l'objet et ces valeurs équivalentes prises dans un autre objet, qui est passé comme paramètre de cette méthode. Les deux méthodes du prototype *Oscillator* mettent également à jour la valeur globale *globalGraph*.

```
Oscillator = function(eq) {
    this.t = 0;
```

```
    this.x = 0;
    this.y = 0;
    this.i = 3000;
    this.eq = eq;
    this.graph = [];
}

Oscillator.prototype.update = function(sum) {
    globalGraph = [];
    this.graph = [];
    var sumFix = (sum) ? sum  : 0;
    for (var i = 0; i < this.i; i++) {
        this.x = this.eq(i, sumFix).x;
        this.y = this.eq(i, sumFix).y;
        var v = createVector(this.x, this.y);
        this.graph.push(v);
    }
    globalGraph = this.graph.slice();
};

Oscillator.prototype.mix = function(sum, o0, sum0, mR) {
    globalGraph = [];
    this.graph = [];
    for (var i = 0; i < this.i; i++) {
        this.x = lerp(this.eq(i, sum).x, o0.eq(i, sum0).x, mR);
        this.y = lerp(this.eq(i, sum).y, o0.eq(i, sum0).y, mR);
        var v = createVector(this.x, this.y);
        this.graph.push(v);
    }
    globalGraph = this.graph.slice();
}
```

2.2 Déclaration des oscillateurs

Les oscillateurs sont ensuite tous créés selon le même format. Ils envoient au constructeur de l'objet *Oscillator* une valeur qui est une fonction, qui devient ensuite la valeur *eq*.

```
var formuleMagique = new Oscillator(function(i, sum) {
    var m = sin((drawCount-sum)/20)/100;
    t = i/30;
    return {
        x : cos(t) * sin(t/2) * (sin(t*(2+m))/4) * 2400 * i/1000,
        y : pow(sin(t/2),3) * cos(t/20) * 350
    };
});
```

2.3 Timing et x-sheets

Pour construire un film entier à partir de scènes différentes où l'on voit une variété d'oscillateurs, il me fallait un système flexible. Il me fallait être capable d'ajouter, d'enlever, de raccourcir et d'allonger des scènes sans affecter les autres. Il était donc impensable que je construise le film entier à partir d'opérateurs *if* et de la valeur *frameCount*. Je me suis donc construit un système qui ressemble aux x-sheets utilisées en animation traditionnelle.

xSheet est un objet littéral qui contient une grande quantité d'objets. Chaque sous-objet est une *scène* du film. La valeur *d* est la durée (ou *duration*) de la scène, et la valeur *f*, la fonction qui est lancée lorsque cette scène roule.

```
var xSheet = {
    pasMal : {
        d :  300,
        f :  function(sum){
                var coFade = cosineFade(sum, 40);
                pasMalCool.update( );
            }
    },
    spiToE : {
        d :  600,
        f :  function(sum){
                var rS = getSum(xSheet,xSheet.spiToE);
                var coFade = cosineFade(sum, 500);
                spiraToupEntre.mix(rS+250, pasMalCool, 0, coFade);
            }
    },
    intere : {
        d :  600,
        f :  function(sum){
                var rS = getSum(xSheet,xSheet.spiToE);
                var coFade = cosineFade(sum, 300);
                interessant.mix(0, spiraToupEntre, rS+250, coFade);
            }
    },
    key : function(n) {
        return this[Object.keys(this)[n]];
    }
};
```

Également, l'objet xSheet a besoin de quelques fonctions extérieures pour fonctionner. Ces fonctions pourraient devenir des méthodes de l'objet lui-même dans le futur, mais pour l'instant ça fonctionne bien comme ça.

```
Object.size = function(obj) {
    var size = 0, key;
    for (key in obj) {
        if (obj.hasOwnProperty(key)) size++;
    }
    return size;
```

```
};

function cosineFade(sum, dur) {
    var fade = map(drawCount,sum,sum+dur,1,0);
    var fadeCons = constrain(fade,0,1);
    var fadeSmooth = fadeCons*PI;
    var coFade = map(cos(fadeSmooth),1,-1,0,1);
    return coFade;
}
```

2.4 Fonctions draw, runXSheet et getSum

La fonction *draw* est ensuite utilisée pour faire rouler la fonction *runXSheet*, qui consulte en boucle l'objet *xSheet* et en active les fonctions appropriées selon la valeur *drawCount*. À l'avenir, il me faudra utiliser ce même système de x-sheets pour faire rouler différents objets du prototype *Postprocessor*. Pour l'instant je n'ai pas assez de tels objets pour que ça vaille la peine.

```
function draw() {
    background(0);
    runXSheet(xSheet);
    simple.output(globalGraph,1);
    if (exporting) {frameExport();}
    drawCount++;
}

function runXSheet(sheet) {
    var tL = Object.size(sheet);

    if (drawCount < sheet.key(0).d) {
            sheet.key(0).f();
    } else {
        for (var i = 1; i < tL; i++) {
            var sum = 0;
            for (var ii = 0; ii < i; ii++){
                sum += sheet.key(ii).d;
            }

            if (drawCount >= sum && drawCount < sum + sheet.key(i).d) {
                sheet.key(i).f(sum);
            }
        }
    }
}

function getSum(sheet, prop) {
    var tL = Object.size(sheet);
    var propLocation = 0;
    var sum = 0;
    for (var i = 0; i < tL; i++) {
        if (sheet.key(i) === prop) {
            propLocation = i;
        }
```

```
    }
    for (var ii = 0; ii < propLocation; ii++){
        sum += sheet.key(ii).d;
    }
    return sum;
}
```

3 Mathématiques pour la simulation d'une troisième dimension

Les points sont générés avec un *for loop* qui incrémente une valeur i. Les valeurs sont multipliées par un quotient de la valeur i, par exemple $\frac{i}{1000}$. Donc il est normal que les points deviennent de plus en plus éloignés à mesure que le graphe se construit, puisque la valeur i augmente.

Ensuite, j'inclus une valeur m dans mon équation, définie ainsi :

```
dC = sin(frameCount/20)/100;
```

Voici la première équation paramétrique avec laquelle j'ai utilisé la valeur m :

$$
\begin{aligned}
t &= \frac{i}{10} \\
x &= \cos(t) \times \sin\left(\frac{t}{2}\right) \times \left(\frac{\sin\left(t \times (2+m)\right)}{4}\right) \times 2400 \times \frac{i}{1000} \\
y &= \sin^3\left(\frac{t}{2}\right) \times \cos\left(\frac{t}{20}\right) \times 350
\end{aligned}
$$

```
t = i/10;
x = cos(t) * sin(t/2) * (sin(t*(2+m))/4) * 2400 * i/1000;
y = pow(sin(t/2),3) * cos(t/20) * 350;
```

4 Algorithmes de création de sommets

J'ai déjà développé des dizaines d'algorithmes différents pour créer mes graphes de façons diverses.

4.1 Méthode de déclanchement des algorithmes

Tout d'abord, dans ma fonction draw, je dessine un background pour effacer le frame précédent, puis je vide l'array vertices. Je crée ensuite la valeur dC (drawCount), qui est pour l'instant plutôt mal définie. Et il en existe 2 versions. La première utilisée est maintenant mise en commentaire. Ensuite, je lance la fonction createVertices() avec la variable dC comme paramètre. L'idée de base était d'introduire une variation à chaque lancement de createVertices, puisqu'ensuite, createVertices fait toujours la même chose, c'est une suite finie et non-ambigüe d'instructions (donc un algorithme).

Ensuite, une fois que le graphe vertices est rempli, je lance la fonction drawVerticesSimple(), qui pour l'instant ne fait que looper dans l'array et dessiner une ellipse blanche pour chaque sommet.

```
function draw() {
  background(0);
  vertices = [];
  // dC = sin(frameCount/20)/100;
  dC = sin(frameCount/700)/5;
  createVertices(dC);
  drawVerticesSimple();
}
```

4.2 La fonction drawVertices()

Cette fonction est très simple. Elle a un paramètre m, et a besoin d'une équation paramétrique qui lui donne des valeurs x et y afin de créer des vecteurs $\vec{v}$. Elle ajoute ensuite ces vecteurs dans l'array *vertices*.

```
function createVertices(m) {
    for (var i = 0; i < 3000; i++) {
        //Insérez une équation paramétrique ici.
        var v = createVector(x, y);
        vertices.push(v);
    }
}
```

5 Équations paramétriques

Voici la liste de toutes les équations dont je me sert pour créer *La nuit géométrique*. Le nombre d'itérations est indiqué à titre de simple suggestion.

5.1 Formule magique

C'est ma toute première équation qui utilise la valeur m.

```
t = i/10;
x = cos(t) * sin(t/2) * (sin(t*(2+m))/4) * 2400 * i/1000;
y = pow(sin(t/2),3) * cos(t/20) * 350;
```

5.1.1 Formule magique modifiée

```
t = i/10;
x = sin(t*(2+m)) * 400 * i/1000;
y = sin(t*(−2+m)) * 350;
```

5.1.2 Spirale en toupie horizontale I

Cette formule a d'abord été créée avec une valeur i maximale de 1000, et un valeur m générée ainsi :

```
dC = sin(frameCount/20)/100;
```

$$
\begin{aligned}
t &= \frac{i}{10} \\
x &= \sin\big(t \times (2+m)\big) \times 400 \times \frac{i}{1000} \\
y &= \cos\big(t \times (2+m)\big) \times \sin\big(t \times (4 \times m)\big) \times 350
\end{aligned}
$$

```
t = i/10;
x = sin(t*(2+m)) * 400 * i/1000;
y = cos(t*(2+m)) * sin(t*(4*m)) * 350;
```

5.1.3 Spirale extra-magique 2

1000 itérations.

```
t = i/10;
x = sin(t*(2+m)) * sin(t*(2*m)) * 400 * i/1000;
y = cos(t*(2+m)) * sin(t*(2*m)) * 350;
```

5.1.4 Juste débile

```
t = i/10;
x = sin(t*(2+m)) * sin(t*(m/2)) * 800 * i/1000;
y = cos(t*(2+m)) * sin(t*(2*m)) * 350;
```

5.1.5 Formule modifiée - Intéressant !

```
t = i/10;
x = sin(t*(2+m)) * cos(t*(4*m)) * 800 * i/1000;
y = cos(t*(2+m)) * cos(t*(2*m)) * 350;
```

5.1.6 Spirale en toupies entrelacées

```
dC = sin(frameCount/700)/5;
```

$$
\begin{aligned}
t &= \frac{i}{10} \\
x &= \sin\left(t \times (2+m)\right) \times \cos\left(t \times \frac{m}{4}\right) \times 800 \times \frac{i}{1000} \\
y &= \cos\left(t \times (2+m)\right) \times \cos\left(t \times \frac{m}{4}\right) \times 350
\end{aligned}
$$

```
t = i/10;
x = sin(t*(2+m)) * cos(t*(m/4)) * 800 * i/1000;
y = cos(t*(2+m)) * cos(t*(m/4)) * 350;
```

5.1.7 Spirale envoûtante

```
t = i/10;
x = sin(t*(2+m)) * asin(t*(m/4)) * 800 * i/1000;
y = cos(t*(2+m)) * asin(t*(m/4)) * 350;
```

6 Équations paramétriques en alternance

Je me sert d'une fonction oscillante pour modifier certains de mes graphes. Mon oscillation est générée ainsi : je me crée une valeur oscillante entre 0 et 1 à partir de la fonction sinus du frameCount, que je "normalise" avec la fonction map. Ensuite, je fais une interpolation linéaire entre deux équations paramétriques, en utilisant mon oscillateur comme valeur d'interpolation. J'obtiens ainsi un graphe qui oscille entre deux équations paramétriques. Chaque sommet du graphe semble déchiré entre 2 objectifs, 2 forces différentes qui le poussent, et l'oscillation est douce puisque j'utilise la fonction sinus.

J'ai conçu mentalement cette notion d'oscillateur un peu à la manière d'un vocoder, c'est-à-dire qu'un vocoder a besoin de deux choses : un modulateur (la voix humaine) et un *carrier signal* ou *onde porteuse*. Dans mon esprit, la première équation paramétrique est le modulateur, et la deuxième équation est l'onde porteuse. Mais au fond, les deux équations sont tout à fait interchangeables et c'est la fonction d'interpolation linéaire qui détermine l'*expression* de chacune d'elles.

Comme on peut le voir, je me sers des valeurs tt, xx et yy pour les valeurs de l'équation no. 2, et ensuite je fait une interpolation linéaire entre x et xx, puis entre y et yy.

```
//MODULATOR
//OscNorm max  : 0.05;
// t = i/2;
// x = cos(t+(m*10)) * 350 * t/1000;
// y = sin(t+(m*40)) * 350;

//OscNorm max  : 1;
t = i/10;
x = sin(t*(2+m)) * asin(t*(m/20)) * 800 * i/1000;
y = cos(t*(2+m)) * asin(t*(m/20)) * 350;

//CARRIER SIGNAL
//Spirale ultra-magique.
tt = i/10;
xx = sin(t*(2+m)) * 400 * i/1000;
yy = cos(t*(2+m)) * sin(t*(4*m)) * 350;

// tt = i/10;
```

```
// xx = sin(t*(2+m)) * asin(t*(m/20)) * 800 * i/1000;
// yy = cos(t*(2+m)) * asin(t*(m/20)) * 350;

//OSCILLATOR FUNCTION
oscillator = sin(frameCount/20);
oscNorm = map(oscillator, -1, 1, 0, 1);
morphX = lerp(x, xx, oscNorm);
morphY = lerp(y, yy, oscNorm);
x = morphX;
y = morphY;
```

7 Coloration et effets spéciaux

Je vais devoir explorer différents concepts de coloration et d'effets spéciaux.