TP: Géométrie 2D

R5.A.06: Sensibilisation à la programmation multimédia

Abdelkader Gouaïch

2023/2024

Modélisation de la Géométrie Euclidienne Analytique

Remarque :

Pour des raisons pédagogiques, nous présenterons du code écrit en TypeScript. L'avantage de TypeScript réside dans sa capacité à attribuer des types statiques aux valeurs, rendant le code plus explicite et facilitant ainsi notre démarche de modélisation. Dans la suite, nous continuerons à utiliser du JavaScript généré à partir de notre code TypeScript.

Modélisation des Concepts Élémentaires

Nous définissons les trois concepts fondamentaux de la géométrie sous forme de classes abstraites : le point, la ligne et le plan. Nous modélisons également la relation d'incidence entre ces éléments.

Point

```
// Notion de Point Abstrait
abstract class AbstractPoint {
  abstract getCoordinates(): number[];
}
```

Un point est un élément abstrait. A ce niveau de généralité, nous pouvons juste demander à avoir des coordonnées pour un point données.

Ligne

```
// Notion de Ligne Abstraite
abstract class AbstractLine {
  // Relation d'incidence avec un point abstrait
```

```
  abstract isIncidentWith(point: AbstractPoint): boolean;
}
```

Une ligne est un ensemble de points ayant une relation de co-linéarité comme contrainte. Nous pouvons demander à savoir si un point est incident avec une ligne.

Plan

```
// Notion de Plan Abstrait
abstract class AbstractPlane {
  // Relation d'incidence avec une ligne abstraite
  abstract isIncidentWith(line: AbstractLine): boolean;
  // Relation d'incidence avec un point abstrait
  abstract isIncidentWith(point: AbstractPoint): boolean;
}
```

Un plan va contenir des lignes ayant une relation de co-planarité comme contrainte. Nous pouvons demander à savoir si une ligne est incidente avec un plan donné.

```
// Interface pour une transformation géométrique
interface GeometricTransformation {
  transform(point: AbstractPoint): AbstractPoint;
}
```

Une transformation géométrique est une fonction qui produit un point à partir d'un autre. Nous modélisons cette opération à l'aide d'une interface TypeScript.

Espace vectoriel

```
// Notion de Vecteur
class Vector {
  constructor(public coordinates: number[]) {}

  // Multiplication par un scalaire
  scalarMultiply(s: number): Vector {
    return new Vector(this.coordinates.map((x) => x * s));
  }
  // Addition de vecteurs
  add(v: Vector): Vector {
    return new Vector(
      this.coordinates.map((value, index) => value + v.coordinates[index])
    );
  }
  // Soustraction de vecteurs
  subtract(v: Vector): Vector {
    return new Vector(
```

```
      this.coordinates.map((value, index) => value - v.coordinates[index])
    );
  }
}
```

La classe `Vector` ci-dessus modélise un vecteur en tant qu'élément d'un espace vectoriel :

- Elle permet de décrire un vecteur par ses coordonnées scalaires qui doivent être interprétées dans une base de l'espace vectoriel.
- Elle offre des méthodes pour définir de nouveaux vecteurs par multiplication scalaire, addition et soustraction de vecteurs.

```
// Notion d'Espace Vectoriel
class VectorSpace {
  base: Vector[];

  constructor(baseVectors: Vector[]) {
    this.base = baseVectors;
  }
  // Propriété pour obtenir la dimension de l'espace vectoriel
  get dimension(): number {
    return this.base.length;
  }
  // Combinaison linéaire pour créer un vecteur à partir de la base
  linearCombination(coefficients: number[]): Vector {
    let result = new Vector(new Array(this.dimension).fill(0));
    for (let i = 0; i < this.dimension; i++) {
      result = result.add(this.base[i].scalarMultiply(coefficients[i]));
    }
    return result;
  }
}
```

Pour représenter un espace vectoriel, nous avons conçu la classe `VectorSpace`. Nous définissons une base: un ensemble minimal de vecteurs qui génère tous les autres vecteurs de cet espace.

La cardinalité de cet ensemble détermine la dimension de l'espace vectoriel.

Nous supposons aussi que la base est composée de vecteurs linéairement indépendants, une condition que nous présumons être déjà validée par l'utilisateur.

La méthode `linearCombination` permet de générer des vecteurs en présentant un tableau de coefficients, qui sont combinés linéairement avec les vecteurs de la base pour produire un nouveau vecteur. Cela donne un sens à la notion de coordonnés d'un vecteur.

Changement de base

Un vecteur peut être identifié de manière unique par ses coordonnées dans une base d'un espace vectoriel donnée.

Les mêmes coordonnées peuvent représenter un vecteur différent si la base est changée. Ce ne sont pas les coordonnées qui définissent un vecteur, mais plutôt la combinaison linéaire des coordonnées avec les vecteurs de la base.

Une question se pose alors : est-il possible de convertir automatiquement les coordonnées d'un vecteur d'une base B vers une autre base B' ?

Cela est réalisable car il s'agit simplement de réécrire les vecteurs de la base B en termes des vecteurs de la base B'.

$$\vec{v} = x_0\vec{e_0} + x_1\vec{e_1} + ... + x_n\vec{e_n} = x'_0\vec{e'_0} + x'_1\vec{e'_1} + ... + x'_n\vec{e'_n}$$

Il est à noter que les vecteurs $\vec{e_0}, ..., \vec{e_n}$ peuvent être exprimés dans la base B'.

Par exemple, le vecteur $\vec{e_0}$ aura pour coordonnées dans B' :

$$\vec{e_0} = a_{0,0}\vec{e'_0} + a_{0,1}\vec{e'_1} + ... + a_{0,n}\vec{e'_n}$$

Et ainsi de suite pour les autres vecteurs de la base B.

En remplaçant chaque vecteur $\vec{e_i}$ par son expression dans la base B' et en regroupant les termes, nous obtenons une matrice de passage P, dont les colonnes sont les coordonnées des vecteurs $\vec{e_i}$ exprimées dans la base B'.

La matrice de passage permet de convertir les coordonnées d'un vecteur de la base originale B vers celles de la base B' :

$$P[x_0, ..., x_n] = [x'_0, ..., x'_n]$$

La matrice P, appelée matrice de passage, facilite le calcul du changement de base car elle permet de retrouver les nouvelles coordonnées par une simple multiplication matricielle.

Pour calculer cette matrice, il est essentiel de choisir une base de référence commune pour exprimer les coordonnées des vecteurs des deux bases. Une base canonique orthonormée est souvent choisie pour sa simplicité. Les vecteurs de cette base ont exactement un coefficient qui vaut 1 et tous les autres sont nuls.

Construction de la matrice de passage

La matrice de passage est construite en résolvant le système linéaire $Ax_i = b_i$, où A et b_i sont connus, et x_i sont les inconnus. Il s'agit des coordonnées du vecteur $\vec{e'_i}$ dans la base originale.

Nous pouvons utiliser la méthode de Gauss-Jordan pour résoudre ce système d'équations. Cette méthode a été implémentée dans la fonction `solveLinearSystem(A,b)` avec `A` une matrice et `b` un vecteur. Elle retourne le vecteur, `x` qui solution de $Ax = b$

```
// Changement de base
changeBase(fromBase: VectorSpace): number[][] {
  const changeOfBaseMatrix: number[][] = [];
  // Pour chaque vecteur de la nouvelle base
  for (const baseVector of fromBase.base) {
    // Construire le vecteur b du système Ax = b
    const b: number[] = baseVector.coordinates;
    // Construire la matrice A du système Ax = b
    const A: number[][] = this.base.map(u => u.coordinates);
    // Résoudre le système d'équations linéaires Ax = b pour trouver x
    const x: number[] = solveLinearSystem(A, b);
    // Ajouter x comme nouvelle colonne à la matrice de changement de base
    changeOfBaseMatrix.push(x);
  }
  return changeOfBaseMatrix;
}
```

La Géométrie Euclidienne dans le plan

Point2D et Vecteur2D

Nous allons maintenant nous intéresser à la géométrie Euclidienne du plan (en 2 dimension donc). Pour cela nous allons rendre les classes abstraites vues précédemment un peu plus concrètes.

```
// Notion de Point dans le Plan 2D
class Point2D extends AbstractPoint {
  constructor(public x: number, public y: number) {
    super();
  }
  getDimensions(): number {
    return 2;
  }
  getCoordinates(): number[] {
    return [this.x, this.y];
  }
  toVector(): Euclidean2DVector {
    return new Euclidean2DVector(this.x, this.y);
  }
}
```

La première concrétisation concerne le point avec la classe `Point2D`. Un point

du plan se caractérise par deux coordonnées qui vont permettre de lui associer un vecteur qui va le désigner depuis l'origine.

```
// Vecteur dans le Plan Euclidien
class Euclidean2DVector extends Vector {
  constructor(public x: number, public y: number) {
    super([x, y]);
  }
  // Calcul de la norme du vecteur
  norm(): number {
    return Math.sqrt(
      this.coordinates[0] * this.coordinates[0] +
        this.coordinates[1] * this.coordinates[1]
    );
  }

  // Calcul du produit scalaire
  dotProduct(v: Euclidean2DVector): number {
    return (
      this.coordinates[0] * v.coordinates[0] +
      this.coordinates[1] * v.coordinates[1]
    );
  }
}
```

La notion de vecteur dans un plan Euclidien apporte de nouvelles fonctionnalités. Nous pouvons réaliser un produit scalaire avec `dotProduct` et calculer la magnitude du vecteur avec `norm`.

Positionnement dans l'espace à partir de coordonnées

☞ **Note importante :** Le répertoire de travail pour cette section est `etape1`.

Notre objectif est de convertir les coordonnées vers le point correspondant dans le plan.

Il est important de rappeler que nous opérons dans le cadre général de l'espace vectoriel et que les coordonnées doivent être interprétées dans ce contexte, relativement à une base spécifique.

La fonction `drawScene(vspace)` (définie dans `src/step1.js`) reçoit un espace vectoriel en paramètre et dessine une scène en utilisant les coordonnées présentes dans la variable *globale* `coordinates`.

En lançant la page `drawTriangle.html` vous allez remarquer deux sliders en haut à droite qui permettent de modifier les vecteurs de la base de l'espace vectoriel. À chaque modification, la scène sera redessinée pour refléter les nouveaux vecteurs de la base en appelant la fonction `drawScene`.

✎ **A faire :**

- Votre mission consiste à ajuster la fonction `drawScene` pour positionner les points en fonction de leurs coordonnées dans la base.
- Pour cela, vous devez revenir à la définition des coordonnées : elles représentent les coefficients appliqués aux vecteurs de la base pour obtenir une combinaison linéaire qui définit une position dans l'espace à partir de l'origine.
- Pour vérifier que votre code est correct, modifiez la base à l'aide des sliders; vous devriez observer la transformation de la figure géométrique et cela sans altérer ses coordonnées initiales.

Questions :

- Que se produit-il si un vecteur de la base est nul ?
- Que se produit-il si les deux vecteurs de la base sont colinéaires ? (par exemple, $[0, 1]$ et $[0, -1]$)

Le changement de base

☞ **Note importante :** Le répertoire de travail pour cette section est `etape2`.

Notre objectif est de pouvoir modifier notre base de vecteurs (tout en conservant le même point d'origine) et de calculer automatiquement les coordonnées de nos points dans cette nouvelle base.

Il est important de noter que la position d'un point dans le plan est indépendante du choix de la base. Nous devrions toujours localiser nos points aux mêmes emplacements dans le plan si nous changeons de base à condition évidemment de recalculer leurs nouvelles coordonnées.

Par exemple, si nous dessinons une figure géométrique et recalculons les coordonnées des points à chaque changement de base, la figure devrait rester inchangée (contrairement à l'exercice de l'étape précédente).

✎ **A faire :**

- Ouvrez le fichier `drawTriangle.html`. Cette page présente une figure géométrique, un triangle, au center. Nous avons des sliders en haut à droite pour modifier nos vecteurs de la base. En bas, à gauche nous présentons les coordonnées des trois points du triangle.
- Ouvrez le fichier `src/step2.js` consulter le code de la fonction `updateScene`

```
function updateScene() {
  var coordinates = transformation_coordonnee(
    ev_base,
    ev_nouveau,
    coordonees_base
  );
```

```
  drawScene(espace_vectoriel_nouveau, coordinates);
  for (let i = 0; i < coordinates.length; i = i + 2) {
    updateCell("cx" + i / 2, coordinates[i]);
    updateCell("cy" + i / 2, coordinates[i + 1]);
  }
}
```

La fonction `updateScene` est responsable de dessiner le triangle et de mettre à jour l'interface graphique.

Cette fonction sera appelée à chaque modification des vecteurs de la base par les sliders.

Avant de réaliser le dessin, nous appelons la fonction `transformation_coordonnee` qui permet de convertir les coordonnées `coordonees_base` exprimés dans la base `ev_base` en des coordonnées exprimés dans la base `ev_nouveau`.

✎ **A faire :**

- Modifiez la fonction `transformation_coordonnee` dans le fichier `main.js` pour retourner les nouvelles coordonnées
  - créer la matrice de passage en utilisant la fonction `changeBase` de l'objet `espace_vectoriel_nouveau`
  - réaliser le produit matriciel sur tous les points
  - retourner le tableau des nouvelles coordonnées

Transformations en Géométrie Euclidienne

Nous allons désormais explorer les transformations géométriques élémentaires dans le plan 2D.

La géométrie analytique nous offre la possibilité d'effectuer des transformations géométriques à l'aide d'outils d'algèbre linéaire. Chaque transformation géométrique correspond donc à une procédure de calcul algébrique impliquant des additions et des multiplications de scalaires, vecteurs et matrices.

Ces opérations peuvent être réalisées :

- au niveau du CPU, via la programmation en JavaScript ;
- au niveau du GPU, à l'aide de shaders.

Bien que les opérations soient conceptuellement identiques que ce soit au niveau du CPU ou du GPU, la différence fondamentale réside dans la performance. Les opérations algébriques effectuées par les shaders sont parallélisées sur tous les points, tandis que celles effectuées sur le CPU sont séquentielles, traitant les points un par un.

Translation

La translation est la première transformation géométrique que nous étudierons. Elle représente le déplacement d'un point le long d'un vecteur défini.

Spécification Mathématique

Mathématiquement, une translation le long d'un vecteur $\vec{t} = (t*x, t_y)$ est définie par l'application :

$$p = (x, y) \mapsto Tr * \vec{t}(p) = (x', y') \text{ avec } \begin{cases} x' & = & x + t_x \\ y' & = & y + t_y \end{cases}$$

Implémentation Côté CPU

Information : Le répertoire de travail pour cette section est `etape3`.

Le projet dans le répertoire `step3` est configuré pour afficher un rectangle en utilisant deux triangles.

Le code se trouve dans le fichier `./src/main.js`, que vous êtes encouragés à examiner.

Dans ce fichier, la variable globale `translate` représente notre vecteur de translation. Cette variable peut être modifiée par deux curseurs situés en haut à droite de la page HTML. Chaque modification déclenche un appel à la fonction `updateScene`, qui redessine la scène entière.

Vous noterez une modification dans le vertex shader :

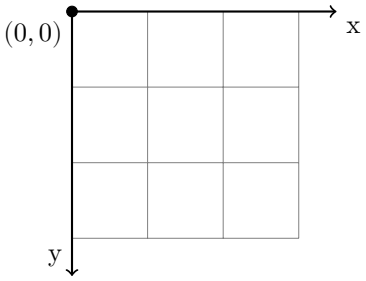
```
attribute vec2 a_position;
uniform vec2 u_resolution;
void main() {
  vec2 zeroToOne = a_position / u_resolution;
  vec2 zeroToTwo = zeroToOne * 2.0;
  vec2 clipSpace = zeroToTwo - 1.0;
  gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
```

Dans les exemples précédents, nous avons utilisé directement les coordonnées de l'espace de dessin (clip space) pour nos points. Les coordonnées de cet espace sont définies entre -1 et 1 pour les deux axes. Nous souhaitons désormais définir nos coordonnées en fonction de la largeur et de la hauteur du canevas de dessin. Nos points sont exprimés dans ce système et doivent être convertis dans l'espace de découpage avant le rendu.

Pour cela, notre shader effectue les opérations suivantes :

- Il calcule une proportion de nos coordonnées par rapport à la largeur et à la hauteur du canevas, résultant en des coordonnées entre 0 et 1 sur les deux axes.

- Il multiplie ces proportions par deux, aboutissant à des coordonnées entre 0 et 2.
- Il soustrait 1 pour obtenir des coordonnées entre -1 et 1.
- Il multiplie par $(1, -1)$ pour ajuster l'orientation de l'axe vertical de sorte que le point $(0, 0)$ soit en haut à gauche et que l'axe vertical soit dirigé vers le bas.



À faire :

- Vous remarquerez que la modification du vecteur de translation via les curseurs n'affecte pas le rectangle.
- Modifiez la fonction `updateScene` pour effectuer la translation du rectangle. Utilisez la fonction `fillRectangleCoordinates(gl, x, y, width, height)` qui construit tous les points d'un rectangle à partir de `x` et `y`, avec une largeur `width` et une hauteur `height`.

Implémentation Côté GPU

☞ **Note importante :** Le répertoire de travail pour cette section est `etape4`.

Notre but est de réaliser la translation point par point au niveau des shaders. Pour cela, nous devons ajouter une nouvelle variable uniforme de type `vec2` dans notre programme de vertex shader pour représenter le vecteur de translation.

```
attribute vec2 a_position;
uniform vec2 u_resolution;
// Nouvelle uniforme pour la translation
uniform vec2 u_translation;
void main() {
  // TODO: prendre en compte la translation de la position : p + t
  vec2 position = a_position;
  // ...
  vec2 zeroToOne = position / u_resolution;
  vec2 zeroToTwo = zeroToOne * 2.0;
  vec2 clipSpace = zeroToTwo - 1.0;
  gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
```

La variable uniforme `u_translation` sera renseignée depuis le CPU avec le code

JavaScript.

Nous procéderons en deux étapes :

- Récupérer l'emplacement de la variable uniforme dans `setupGL()` avec `translationLocation = gl.getUniformLocation(program, "u_translation");`
- Assigner la valeur de la variable uniforme dans `updateScene()` avec `gl.uniform2fv(translationLocation, translate);`

✎ **A faire :**

- Modifiez le code du vertex shader pour effectuer la translation des points.
- Pour vérifier que votre code est correct, vous devriez observer le déplacement de la figure géométrique en modifiant les curseurs.

Rotation

Spécifications Mathématiques

Pour effectuer une rotation R_θ d'un angle θ autour de l'origine, nous appliquons la fonction suivante :

$$p = (x, y) \mapsto R_\theta(p) = (x', y') \text{ avec } \begin{cases} x' &= x \cdot \cos(\theta) - y \cdot \sin(\theta) \\ y' &= x \cdot \sin(\theta) + y \cdot \cos(\theta) \end{cases}$$

Implémentation

☞ **Note importante :** Le répertoire de travail pour cette section est `etape5`.

```
attribute vec2 a_position;
uniform vec2 u_resolution;
uniform vec2 u_translation;
// Nouveau
uniform vec2 u_rotation;
//

void main() {
  // TODO
  // Calculer la nouvelle position en tenant compte
  // de la rotation
  vec2 rotatedPos = vec2(a_position.x, a_position.y);
  // ...
  // Phase de translation
  vec2 position = rotatedPos + u_translation;
  //Phase de projection
  vec2 zeroToOne = position / u_resolution;
  vec2 zeroToTwo = zeroToOne * 2.0;
  vec2 clipSpace = zeroToTwo - 1.0;
```

```
  gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
```

Ce vertex shader actuel ne calcule pas de rotation ; il se contente de reproduire les valeurs de l'attribut `a_position`.

✎ **A faire :**

- Modifiez le code de ce shader pour effectuer une rotation du point, sachant que l'uniform `u_rotation` contient le sinus et le cosinus de l'angle de rotation dans `u_rotation.x` et `u_rotation.y`, respectivement.
- Notez que nous effectuons une rotation autour de l'origine, puis une translation. L'ordre de ces opérations est-il important ?

Dilatation

Spécifications Mathématiques

Pour effectuer une dilatation H_{s_x,s_y} de facteurs s_x et s_y selon les axes $\vec{x}$ et $\vec{y}$, nous réalisons la transformation suivante :

$$p = (x, y) \mapsto H_{s_x,s_y}(p) = (x', y') \text{ avec } \begin{cases} x' &= s_x \cdot x \\ y' &= s_y \cdot y \end{cases}$$

Implémentation

☞ **Note importante :** Le répertoire de travail pour cette section est `etape6`.

✎ **A faire :**

En suivant les mêmes étapes que précédemment, modifiez le vertex shader dans `src/main.js` pour gérer la dilatation, en tenant compte des éléments suivants :

- Le script `main.js` déclare déjà une variable `let scale = [1,1];`.
- La page HTML définit deux curseurs supplémentaires qui modifient la variable `scale` et appellent la fonction `updateScene` après chaque changement.

Vous devez réaliser les points suivants :

- Déclarer une uniforme `u_scale` dans le vertex shader.
- Déclarer une variable pour gérer la localisation de l'uniforme `u_scale`.
- Dans la fonction `updateScene`, transférer correctement les valeurs de la variable `scale` vers l'uniforme `u_scale` en utilisant `gl.uniform2fv()`.
- Mettre à jour le vertex shader pour intégrer la transformation de dilatation. L'ordre des transformations sera : dilatation, puis rotation, et enfin translation.

Testez votre application une fois ces opérations réalisées.

La correction de l'orthographe et du style du texte fourni nécessite une attention particulière aux détails techniques ainsi qu'aux conventions de la notation mathématique et de la programmation. Voici le texte corrigé avec les modifications nécessaires :

Cadre général des transformations à l'aide de matrices

Revoyons les formules des trois transformations que nous avons précédemment introduites.

$$Tr_{(t_x,t_y)}(p) = (x', y') \text{ avec } \left\{ \begin{array}{rcl} x' & = & x + t_x \\ y' & = & y + t_y \end{array} \right.$$

$$R_\theta(p) = (x', y') \text{ avec } \left\{ \begin{array}{rcl} x' & = & x\cos\theta + y\sin\theta \\ y' & = & y\cos\theta - x\sin\theta \end{array} \right.$$

$$H_{(s_x,s_y)}(p) = (x', y') \text{ avec } \left\{ \begin{array}{rcl} x' & = & xs_x + y \cdot 0 + 1 \cdot 0 \\ y' & = & x \cdot 0 + ys_y + 1 \cdot 0 \end{array} \right.$$

Nous pouvons reformuler ces équations de la manière suivante :

$$Tr_{(t_x,t_y)}(p) = (x', y') \text{ avec } \left\{ \begin{array}{rcl} x' & = & x \cdot 1 + y \cdot 0 + t_x \cdot 1 \\ y' & = & x \cdot 0 + y \cdot 1 + t_y \cdot 1 \end{array} \right.$$

$$R_\theta(p) = (x', y') \text{ avec } \left\{ \begin{array}{rcl} x' & = & x\cos\theta + y\sin\theta + 0 \cdot 1 \\ y' & = & y\cos\theta - x\sin\theta + 0 \cdot 1 \end{array} \right.$$

$$H_{(s_x,s_y)}(p) = (x', y') \text{ avec } \left\{ \begin{array}{rcl} x' & = & xs_x + y \cdot 0 + 0 \cdot 1 \\ y' & = & x \cdot 0 + ys_y + 0 \cdot 1 \end{array} \right.$$

Nous observons un schéma récurrent dans ces systèmes d'équations :

$$\text{avec } \left\{ \begin{array}{rcl} x' & = & xa_{00} + ya_{01} + a_{02} \\ y' & = & xa_{10} + ya_{11} + a_{12} \end{array} \right.$$

Cela nous rappelle les règles de multiplication d'une matrice par un vecteur :

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Nous pouvons alors exprimer toutes les transformations sous la forme d'un produit matriciel. Il suffit de choisir les coefficients appropriés pour réaliser une translation, une rotation ou une homothétie.

Cependant, nous remarquons que nous partons initialement d'un vecteur de dimension 3×1 et obtenons en résultat un vecteur de dimension 2×1.

Ce phénomène s'explique par le fait que nous multiplions une matrice 2×3 par un vecteur 3×1, ce qui donne un vecteur 2×1.

Cela peut poser problème si nous souhaitons appliquer directement une autre transformation sous forme de produit matriciel avec une autre matrice 2×3, car il faudrait modifier notre vecteur 2×1 en lui ajoutant une ligne.

Une solution pratique consiste à partir d'un vecteur de dimension 3×1 et à obtenir un vecteur de même dimension. Pour ce faire, il suffit d'ajouter artificiellement une ligne à notre matrice de transformation, qui aura alors une dimension de 3×3 :

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Appliquons à présent la multiplication matricielle et observons le résultat :

$$\text{avec } \begin{cases} x' &= xa_{00} + ya_{01} + a_{02} \\ y' &= xa_{10} + ya_{11} + a_{12} \\ 1 &= 0 \cdot x + 0 \cdot y + 1 \end{cases}$$

Le résultat est cohérent et nous permet de modéliser une séquence de transformations comme une série de multiplications matricielles.

$$p' = Tr_{(t_x,t_y)} \cdot R_\theta \cdot H_{(s_x,s_y)} \cdot p$$

Les règles du calcul matriciel nous autorisent à réécrire cette équation en effectuant d'abord les produits matriciels, puis en appliquant la transformation au point :

$$p' = Tr_{(t_x,t_y)} \cdot R_\theta \cdot H_{(s_x,s_y)} \cdot p$$

Matrices de transformation

Voici les matrices 3×3 correspondant aux transformations précédemment décrites.

Matrice de Translation

La matrice de transformation pour une translation selon un vecteur (t_x, t_y) est :

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Matrice de Rotation

La matrice de transformation pour une rotation d'un angle θ est :

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Matrice d'Homothétie

La matrice de transformation pour une homothétie de coefficients (s_x, s_y) est :

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Implémentation

☞ **Note importante :** Le répertoire de travail pour cette section est `etape7`.

Nous proposons une bibliothèque pour la gestion des matrices 3×3 sous forme d'un objet `m3`.

Nous allons exposer quelques choix pour l'implémentation de cette bibliothèque. Pour plus de détails, vous pouvez consulter l'implémentation de cet objet dans le fichier `main.js`.

Premièrement, cette bibliothèque est dédiée uniquement aux matrices de dimension 3×3. Nous pouvons donc optimiser certaines opérations, comme le produit de deux matrices de cette dimension.

Nous avons également choisi de représenter une matrice 3×3 par un tableau de neuf éléments organisés par colonnes. Chaque groupe de trois éléments représente donc une colonne de la matrice. Cette représentation nous permet de transférer directement le tableau à la GPU, qui représente également les matrices 3×3 par un tableau de neuf éléments, organisés en colonnes.

La première fonction concerne la multiplication matricielle. `m3` offre une fonction `multiply(a, b)` qui retourne le résultat de la multiplication des deux matrices $b \times a$.

`m3` permet d'obtenir la matrice de translation avec la fonction :

```
translation: function(tx, ty) {
  return [
    1, 0, 0,
    0, 1, 0,
    tx, ty, 1,
  ];
},
```

Vous remarquerez qu'il s'agit de la transcription de notre spécification mathématique. Notez que notre représentation des matrices est par colonne. Les trois premiers éléments représentent la première colonne de notre écriture mathématique, et ainsi de suite.

Pour la rotation :

```
rotation: function(angle) {
  var c = Math.cos(angle);
  var s = Math.sin(angle);
  return [
    c, -s, 0,
    s, c, 0,
    0, 0, 1,
  ];
},
```

Et pour l'homothétie :

```
scaling: function(sx, sy) {
  return [
    sx, 0, 0,
    0, sy, 0,
    0, 0, 1,
  ];
},
```

Au niveau du vertex shader, nous avons uniquement besoin de la matrice de transformation que nous appliquons à la position du vertex :

```
attribute vec2 a_position;
uniform vec2 u_resolution;
//NEW
uniform mat3 u_matrix;
void main() {
  // NEW: appliquer la transformation par multiplication
  vec2 position = (u_matrix * vec3(a_position, 1)).xy;
  //---
  vec2 zeroToOne = position / u_resolution;
  vec2 zeroToTwo = zeroToOne * 2.0;
```

```
  vec2 clipSpace = zeroToTwo - 1.0;
  gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
```

✎ **A faire :**

Modifiez le programme `main.js` afin de calculer une matrice qui représente les différentes transformations, l'homothétie, la rotation et la translation dans cet ordre. Vous devez transférer cette matrice à l'uniform `u_matrix` du vertex shader en utilisant la fonction `gl.uniformMatrix3fv(matrixLocation, false, matrix);`

Voici les détails des étapes à réaliser :

- Récupérez la localisation de l'uniform `u_matrix` et rangez-la dans la variable `matrixLocation`.
- À chaque dessin de la scène, calculez une matrice `matrix` qui sera le produit des matrices suivantes et dans cet ordre : homothétie, rotation, puis translation. Utilisez la bibliothèque `m3` pour réaliser ces opérations.
- Transférez `matrix` à l'uniform `u_matrix` du vertex shader en utilisant la fonction `gl.uniformMatrix3fv(matrixLocation, false, matrix);`

Vous pouvez tester votre programme en jouant avec les curseurs.

Voici une version corrigée et améliorée du texte, avec une attention particulière portée à la clarté, la précision et le style formel :

La Matrice de Projection

☞ **Note importante :** Le répertoire de travail pour cette section est `etape8`.

Examinons le code actuel de notre vertex shader :

```
attribute vec2 a_position;
uniform vec2 u_resolution;
uniform mat3 u_matrix;

void main() {
    // Transformation
    vec2 position = (u_matrix * vec3(a_position, 1)).xy;
    // Projection
    vec2 zeroToOne = position / u_resolution;
    vec2 zeroToTwo = zeroToOne * 2.0;
    vec2 clipSpace = zeroToTwo - 1.0;
    gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
```

Ce traitement se décompose en deux phases distinctes :

1. Une phase de transformation géométrique qui modifie les coordonnées des points en fonction de notre espace vectoriel défini.
2. Une phase de projection sur le cadre de dessin, qui requiert la conversion des coordonnées de l'espace vectoriel des objets vers celui du cadre de dessin, délimité par les valeurs -1 et 1 sur les deux axes.

Il est à noter que les opérations réalisées durant la phase de projection sont purement algébriques :

- Division par la résolution (largeur et hauteur du cadre de dessin).
- Multiplication par 2.
- Soustraction de 1.
- Inversion de l'axe vertical par multiplication de la deuxième composante par -1.

Heureusement, ces opérations peuvent être exécutées via un calcul matriciel. La matrice suivante effectue l'ensemble de ces transformations :

$$\begin{pmatrix} 2/w & 0 & -1 \\ 0 & -2/h & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

En multipliant cette matrice par un vecteur $(x, y, 1)$, nous obtenons :

$$\begin{cases} x' = x \cdot \frac{2}{w} - 1 \\ y' = y \cdot \frac{-2}{h} + 1 \\ 1 = 1 \end{cases}$$

Ce résultat est identique à celui produit par les opérations algébriques de notre shader.

L'approche logique consiste donc à représenter également la phase de projection sur le cadre de dessin à l'aide d'une matrice. La transformation globale s'effectuera alors par un produit matriciel : la matrice de projection multipliée par le vecteur de position du point.

Le code de notre vertex shader se simplifie ainsi :

```
attribute vec2 a_position;
uniform mat3 u_matrix;

void main() {
    // Transformation et projection
    vec2 position = (u_matrix * vec3(a_position, 1)).xy;
    gl_Position = vec4(position, 0, 1);
}
```

✎ **A faire :**

Le vertex shader requiert désormais une matrice qui intègre à la fois les transformations et la projection vers le cadre de dessin. Il convient de modifier le code de la fonction `updateScene` afin de calculer la matrice englobant la phase de projection.

Les étapes à suivre sont les suivantes :

- Intégrer une fonction `projection(w, h)` à l'objet `m3`, qui générera une matrice de projection adaptée à un cadre de largeur `w` et de hauteur `h`.
- Utiliser le produit matriciel pour concaténer la projection aux transformations de dilatation, rotation et translation déjà existantes.
- Transmettre la matrice résultante au vertex shader.

Ces modifications permettront d'assurer une cohérence entre les transformations géométriques et la projection finale sur le cadre de dessin.