Cours 10

Transformations affines

- En mathématique, on dit qu'une transformation affine est l'application d'une translation et une affinité entre deux espaces affines.
- La transformation représente une changement de système de coordonnées.
- Par exemple, une vue à son propre origine, qui n'est pas nécessairement le même que l' écran entier. Pour obtenir les coordonnées locales d'une vue, on effectue une transformation affine aux coordonnées globales de l'écran.

- L’utilisation de transformations affines peut simplifier énormément vos algorithms.
- Par exemple, si nous implémentons un système de collision AABB (*Axis-Aligned Bounding Box*). Ce système est la forme de collision la plus simple, ne prenant en considération que la position et la taille (largeur et hauteur).

- Si notre AABB à cette forme, le code devient:

```
public class AABB
{
    public int x, y, width, height;

    public boolean collision(int x, int y) {
        int dx = x - this.x;
        int dy = y - this.y;
        return dx >= 0 && dx < width && dy > 0 && dy < height;
    }
}
```

- Sans le savoir, la première partie de notre méthode de collision utilise déjà une transformation affine. Ici, nous appliquons une translation pour convertir les coordonnées en coordonnées locales à notre boîte:

```
public class AABB
{
    public int x, y, width, height;

    public boolean collision(int x, int y) {
        int dx = x - this.x;
        int dy = y - this.y;
        return dx >= 0 && dx < width && dy > 0 && dy < height;
    }
}
```

- Par contre, notre boîte est présentement positionné à son coin inférieur. Généralement, il est préférable d'avoir la position au milieu d'un objet pour faciliter les manipulations.

(x + width / 2, y + height / 2)

(x - width / 2, y - height / 2)

```
public class AABB
{
    public int x, y, width, height;

    public boolean collision(int x, int y) {
        int dx = x - this.x;
        int dy = y - this.y;
        return dx >= -width / 2 && dx < width / 2 && dy > -height / 2 && dy < height / 2;
    }
}
```

- Ce code est déjà plus laid et demande un effort supplémentaire à lire. On pourrait donc déplacer cette logique dans notre transformation:

```
public boolean collision(int x, int y) {
    int dx = x - this.x + width / 2;
    int dy = y - this.y + height / 2;
    return dx >= 0 && dx < width && dy > 0 && dy < height;
}
```

- Ou nous pourrions également déplacer entièrement les informations de **`width`** et **`height`** dans la transformation!

```
public class AABB {

    public float x, y, width, height;

    public boolean collision(float x, float y) {
        float dx = (x - this.x) / width + 0.5f;
        float dy = (y - this.y) / height + 0.5f;
        return dx >= 0.0f && dx < 1.0f && dy >= 0.0f && dy < 1.0f;
    }
}
```

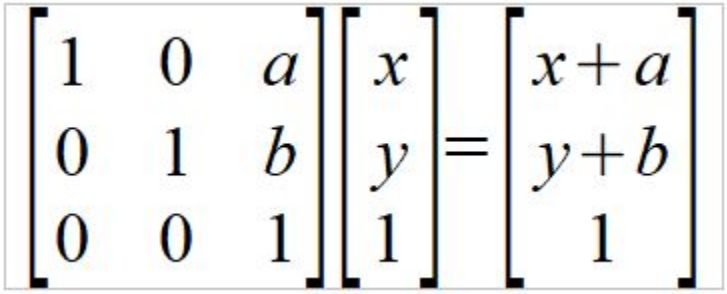
- Sous cette forme, ce que nous considérons réellement est que notre boîte est un carré avec une taille de 1 par 1, ayant subit une translation et un s'étant fait étiré pour prendre une dimension d'un rectangle. Les dimensions sont des "*float*" puisque tout est maintenant une fraction normalisée.
- On remarque que chaque coordonnée est maintenant affecté par une translation et une multiplication. Cette multiplication provient de l'autre composante d'une transformation - l'affinité.

- Il peut sembler complexe de faire ceci pour une simple AABB, mais l'exercice s'applique également à une OOBB (*Object-Oriented Bounding Box*). Implémenter une OOBB est de loin plus complexe puisqu'elle doit supporter l'orientation de l'objet (rotation).
- Par contre, la vérification finale d'une OOBB aurait le même code si la rotation serait également déplacée dans la transformation.

- Ce genre de test de collision pourrait être fait dans Unity avec simplement la dernière ligne. En déplacement la position, la taille et la rotation dans la transformation, elles peuvent maintenant être faites à l'aide de la propriété **`transform`** d'un "*game object*"!
- La propriété **`transform`** d'un "*game object*" est justement le support de transformations affines dans Unity!
- Chaque "*game object*" a son propre espace de coordonnées obtenu en appliquant sa transformation à l'espace de son parent!

- Généralement, les transformations sont faites à l'aide de matrices, et les cartes graphiques ont du matériel spécialisé pour les multiplications rapides de matrices.
- Le faire de cette façon permet de déplacer une partie du travail vers la carte graphique!
- Toutes les coordonnées que vous dessinez sont toujours multipliées par au moins une transformation: la matrice de projection.

- Toutes les transformations communes peuvent s'exprimer sous la forme d'une multiplication de matrices. Par exemple, pour effectuer une translation de **`(a, b)`** au point **`(x, y)`**, on fait ceci:



- La matrice à gauche est une matrice augmentée ayant des dimensions de 1 de plus que le nombre de coordonnées. Également, les deux autres matrices représentent le point avant et après, et sont des colonnes ayant une rangée de plus que le nombre de coordonnées (cette rangée aura toujours la valeur 1).

- On pourrait également faire un “*scale*” de **`a`** fois horizontalement par **`b`** fois verticalement en utilisant:

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax \\ by \\ 1 \end{bmatrix}$$

- On pourrait également les combiner avec:

$$\begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x x + t_x \\ s_y y + t_y \\ 1 \end{bmatrix}$$

- D'ailleurs, toutes les combinaisons de transformations peuvent se faire représenter en une seule matrice! Il suffit de les multiplier ensemble!

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Faites attention! L'ordre est important! Si vous faites le "*scale*" avant la translation, le "*scale*" s'applique à la valeur de la translation!

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & s_x t_x \\ 0 & s_y & s_y t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Sur Android, vous n’aurez pas besoin de connaître comment composer vous-même une matrice de translation, rotation, etc.
- La classe **`Canvas`** a déjà toutes les méthodes nécessaires pour calculer pour vous ces matrices de transformation.
- Le **`Canvas`** garde a toujours une matrice active qui est appliquée à tous les points que vous lui passé.

- Vous utiliserez principalement les méthodes suivantes:
  - **`void rotate(float degrees)`** : Effectue une rotation des axes à l'origine, dans la direction de l'axe X positif vers l'axe Y positif.
  - **`void scale(float sx, float sy)`** : Multiplie l'axe X et l'axe Y par **`sx`** et **`sy`** respectivement. Peut également faire un effet de miroir en utilisant une valeur négative!
  - **`void translate(float dx, float dy)`** : Déplace l'origine au point **`(dx, dy)`**.

- Chacune de ces opérations sont destructives à la matrice actuelle. Si vous voulez pouvoir retourner à la valeur précédente, vous devez utiliser:
  - **`int save()`** : Sauvegarde la valeur actuel à l'ajoutant au sommet de la pile interne.
  - **`void restore()`** : Récupère la valeur actuellement au sommet de la pile.
- Tout appel à **`save()`** devrait toujours être accompagné d'un appel à **`restore()`**!

- Conseils:
  - Lorsque vous dessinez, essayez toujours de garder l'origine au point pivot naturel de votre forme. Ceci est généralement le milieu ou un autre point logique (par exemple, les pieds pour un personnage).
  - Votre forme ne devrait pas avoir besoin de connaître des dimensions externes. Gardez-la simple en utilisant des dimensions constantes et appliquant simplement **`scale`** si nécessaire.
  - Essayez de toujours pointer dans une direction aligné avec l'un des axes. De préférence, utilisez l'axe des X, car il est plus simple de visualiser ainsi une rotation.

Cours 10

Les Dialogues

Les Dialogues

- Les dialogues sont des petites fenêtres affichées pour interagir avec l'utilisateur.
- Ils sont appropriés pour demander une confirmation or clarification pour une action de l'activité actuelle. Ils sont également utilisés pour afficher des erreures ou le résultat d'une opération.
- Contrairement aux activités, un dialogue n'est pas présent sur la pile de tâches. Il est considéré comme faisant partie de l'activité l'ayant instancié, de la même façon que les vues de l'activité.
- Afficher un dialogue ne provoque pas d'appel à **`onPause`** puisqu'il fait partie de l'activité.

- Tous les dialogues sont des instances de sous-classes de **`Dialog`**.
- Il existe plusieurs classes appropriées pour les différents types de dialogues désirés (**`DatePickerDialog`**, **`TimePickerDialog`**, **`ProgressDialog`**, …).
- Généralement, les dialogues sont des instances de **`AlertDialog`** qui est une classe qui s'adapte à toutes les utilisations.

AlertDialog

- La classe **`AlertDialog`** permet d'afficher des fenêtres incluant:
  - un titre;
  - une icône;
  - un message;
  - une liste d'éléments;
  - jusqu'à 3 boutons (réponse négative, positive ou neutre).
- Tous les éléments sont optionnels et l'interface s'adapte aux éléments choisis.

- Pour construire un **`AlertDialog`**, on passe par sa classe imbriquée **`AlertDialog.Builder`**.
- Cette classe permet de créer un dialogue en enchaînant des appels de méthodes (cette forme de programmation se nomme “*fluent programming*”). Une fois le contenu déterminé, on peut simplement appeler la méthode **`show()`** pour afficher le dialogue résultant.

- On comme en instanciant le “*builder*” en l’associant à l’activité dont il fait partie.

```
AlertDialog.Builder builder =  new AlertDialog.Builder( this);
```

- Toutes les méthodes retournent une référence au “*builder*” et peuvent alors être enchaînées sans spécifier ce dernier à chaque ligne.

```
builder.setTitle("Mon titre").setMessage("Mon message");
```

- On peut configurer également le comportement, comme la possibilité d'annuler le dialogue avec la méthode **`setCanceable()`**.
- Pour associer une action à un bouton, on utilise **`setNegativeButton()`**, **`setPositiveButton()`** ou **`setNeutralButton()`**.
- Chacune de ces méthodes prend une instance d'une interface imbriquée de **`DialogInterface`**.
- On peut également gérer les événements à l'aide de méthodes telles que **`setOnDismissListener()`**.

- Si les choix de **`AlertDialog`** ne sont pas suffisants, il permet également de spécifier une vue de votre choix et même un fichier de “*layout*”.
- La méthode **`show()`** permet ultimement de créer le dialogue et de l’afficher.

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Mon titre")
        .setMessage("Mon message")
        .setPositiveButton("OK", new DialogInterface.OnClickListener() {

            @Override
            public void onClick(DialogInterface dialog, int which) {

                Log.d(TAG, "L'utilisateur a appuyé sur OK!");
            }
        })
        .show();
```