Théorie TP6

Afin de réaliser votre premier jeu, il y a certaines choses que l'on doit apprendre. De plus, la théorie sur la programmation orientée objet sera utilisée dans la réalisation de ce jeu. Il est donc important d'être à l'aise avec cette nouvelle façon de faire. Si jamais vous avez besoin d'un petit rafraîchissement : théorie POO ou ici.

Avant de commencer à programmer un jeu vidéo, il faut voir certains concepts qui vont nous aider à réaliser notre premier jeu:

- les enum.
- les dictionnaires;
- les sprites;
- les SpriteList;
- les animations.

La classe Enum

Le premier concept est ce que l'on nomme des états de jeu (game state). De façon simple, un état de jeu c'est une manière de dire que notre jeu est dans un état particulier et que l'exécution devrait faire quelque chose en fonction de cet état. Par exemple, dans le jeu Minecraft, lorsque l'on ouvre l'inventaire, le jeu doit afficher l'inventaire à l'écran. Si l'on ferme l'inventaire, le jeu doit cesser d'afficher cette

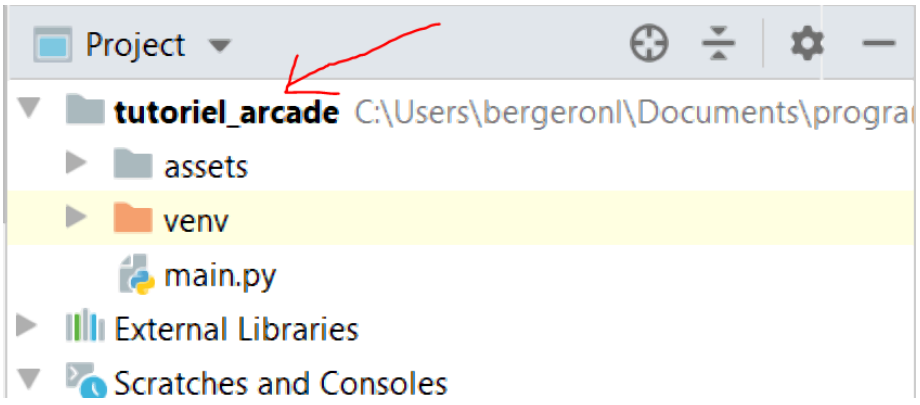
fenêtre. Dans le code responsable du rendering, on pourrait retrouver une structure qui ressemble à cela:

```
if self.game_state == GameState.RenderInventory:
      # Dessiner la fenêtre de l'inventaire
      pass
elif self.game_state == GameState.RenderMenu:
      # Dessiner le menu principal
      pass
```

Comment peut-on faire cela avec Python? La classe Enum!
C'est quoi au juste? La classe Enum permet de faire des énumérations. Selon Antidote, une énumération est "[une] Énonciation successive des éléments d'un tout". Dans un jeu, les états de jeu sont donc une énumération de tous les états possibles que notre jeu peut avoir. Dans les consignes, je vous demande d'en faire 4.

Dans les faits, on va ajouter un nouveau fichier Python à notre projet et nous allons y écrire notre classe GameState.

Pour ajouter un nouveau fichier Python, il faut cliquer avec le bouton droit sur le nom de notre projet dans la section de gauche de PyCharm:



Il faut ensuite sélectionner "New" -> "Python file". Nommer votre nouveau fichier "game_state" et appuyer sur "enter".

Dans ce nouveau fichier, il faut importer la classe Enum de Python. Entrer le code suivant:

```
from enum import Enum
```

```
class GameState(Enum):
   NOT_STARTED = 0
```

Il faut ensuite ajouter les 3 autres états possibles dans ce fichier. Veuillez vous référer au document de consignes pour les connaître.

Dans le jeu, l'état du jeu change en fonction de l'interaction de l'usager. Ainsi, dans l'état `GameState.NOT_STARTED`, le jeu doit attendre que l'usager appuie sur la touche "espace". C'est donc dans la méthode `on_key_press` que l'état va changer.

Les dictionnaires

Un dictionnaire c'est un contenant dont les éléments sont associés à une clé unique. Voici un exemple de l'utilisation d'un dictionnaire avec Python:

```
number_to_text = {
   0: "zéro",
   1: "un",
   2: "deux",
   3: "trois",
   4: "quatre"
}

for i in range(5):
   print(number_to_text[i], end=" ")
```

Si l'on exécute ce petit programme, on obtient le résultat suivant:

```
zéro un deux trois quatre
```

On peut donc voir que si l'on utilise une clé précise, le dictionnaire nous donne la valeur ou l'objet qui y est associé. Pourquoi utiliser ce concept dans le jeu de roche, papier, ciseaux? Si l'on veut savoir quelle est l'attaque du joueur, nous devrions avoir un minimum de 3 variables booléennes. Ça risque de devenir lourd à gérer. Avec un dictionnaire, on peut réduire cela à une seule variable. Mais pour se faire, il faudrait créer un enum pour représenter les types d'attaques. Avec ce dictionnaire, on peut avoir une vérification de ce genre:

```
if self.player_attack_type[AttackType.ROCK]:
     pass
```

C'est très pratique et permet d'avoir un code plus facile à lire.

Pour être en mesure de faire ceci, on doit donc définir `player_attack_type` comme un dictionnaire dans notre classe principale. La clé est le type d'attaque et à chacune de ces clés est associée une valeur booléenne. Voici comment le faire:

```
       self.player_attack_type = {
            AttackType.ROCK: False,
            AttackType.PAPER: False,
            AttackType.SCISSORS: False
        }
```

Les sprites avec Arcade

Un sprite c'est une image que l'on voudrait afficher dans notre programme. La librairie Arcade nous facilite énormément la vie avec sa classe `arcade.Sprite`. Voici les attributs qui nous seront le plus utiles:

- filename : le nom du fichier qui contient votre image;
- scale : l'échelle de votre sprite. 0.5 = moitié moins gros;
- center_x : le centre de votre sprite dans l'écran sur l'axe des X;
- center_y : le centre de votre sprite dans l'écran sur l'axe des Y.

Pour les autres attributs, vous pouvez consulter le site directement.

Il y a deux scénarios possibles pour l'utilisation d'un sprite. Dans un premier temps, si vous avez juste besoin d'afficher un sprite et qu'il n'y a pas d'information attachée à ce dernier, vous allez utiliser la classe `arcade.Sprite` directement:

```
coin = arcade.Sprite(":resources:images/items/coinGold.png", SPRITE_SCALING_COIN)
```

Notez ici que le ":resources:images" est propre à la librairie Arcade. Vous n'allez pas l'utiliser dans votre programme.

Et dans la méthode `on_draw` de votre classe `MyGame`, on écrit le code suivant:

```
coin.center_x = 200
coin.center_y = 200
coin.draw()
```

Vous allez obtenir le résultat suivant:

Donc, pour un simple affichage d'une image, il suffit de créer une variable de type `Sprite` et d'invoquer la méthode `draw()` de celle-ci pour afficher l'image à l'écran.

Une fonctionnalité très intéressante de la classe Sprite qui nous sera très utile est `Sprite.collides_with_point`. Cette méthode nous permet de savoir si un point donné par x et y est à l'intérieur du Sprite. Voici un exemple de code:

```
def on_mouse_press(self, x, y, button, key_modifiers):
   if self.hero.collides_with_point((x, y)):
       print("L'usager a cliqué sur le héros.")
```

C'est donc très utile pour savoir si l'usager clique sur le sprite avec la souris.

Les SpriteList

Pourquoi une SpriteList? Tout simplement parce que cette classe va nous simplifier la vie encore plus. Une classe SpriteList c'est une liste qui contient des Sprites. À première vue, on pourrait simplement utiliser une liste Python pour emmagasiner nos sprites. Ça fonctionnerait. Par contre, une `SpriteList` d'Arcade nous permet de faire beaucoup plus. Premièrement, elle permet d'optimiser l'affichage, la collision de détection ainsi que le mouvement. C'est donc avantageux d'y ajouter les Sprites que l'on veut utiliser dans notre jeu. Même s'il y en a que deux ou trois, c'est une bonne pratique à avoir. Pour l'utiliser, il suffit d'abord de l'ajouter comme attribut dans notre classe `MyGame`:

```
self.coin_list = arcade.SpriteList()
```

Si je veux ajouter 10 items dans ma liste, je procède ainsi :

```
for i in range(10):
   coin = arcade.Sprite(":resources:images/items/coinGold.png")
   self.coin_list.append(coin)
```

Et dans la méthode on_draw de ma classe principale, j'ai juste à faire ceci:

```
self.coin_list.draw()
```

Arcade va donc parcourir la liste et afficher les 10 Sprites qu'elle contient. Elle va aussi maximiser cet affichage pour plus de performance.

Les animations

Chaque type d'attaque sera représenté à l'aide d'une simple animation qui contient deux séquences (frame). Pour faire une animation, il faut créer une nouvelle classe qui hérite des fonctionnalités de `arcade.Sprite`. Si nous avons 3 types d'attaques, devons-nous faire 3 nouvelles classes? La réponse est… non. Nous allons utiliser les Enum pour configurer notre classe. Premièrement, nous allons ajouter un nouveau fichier Python dans notre projet et nous le nommerons "attack_animation". Dans ce fichier, il faut créer un enum pour les types d'attaques.

```
class AttackType(Enum):
    """
    Simple énumération pour représenter les différents types d'attaques.
    """
    ROCK = 0,
    PAPER = 1,
    SCISSORS = 2
```

À partir de cela, il sera possible de configurer quel type d'animation nous voulons créer.

Ensuite, il faut créer notre nouvelle classe:

```
class AttackAnimation(arcade.Sprite):
    ATTACK_SCALE = 0.50
    ANIMATION_SPEED = 5.0
```

Les constantes de la classe seront expliquées ultérieurement.

Il faut donc avoir une méthode `__init__` (dans notre classe AttackAnimation) qui reçoit en paramètre le type d'attaque. Ensuite, nous pouvons charger en mémoire les bons fichiers en fonction du type. Avant d'aller plus loin, je vais vous parler de comment l'on peut animer une séquence de sprite. La classe `arcade.Sprite` contient deux autres attributs pour notre animation:

- texture : un objet de type `Texture` qui représente la texture (ou image) actuellement utilisée pour l'affichage;

- textures : la liste des objets de type `Texture` (ou image) associée au Sprite.

On peut donc revenir à notre "constructeur". Il faut charger les bonnes images en fonction du type d'attaque. Voici le code pour le faire:

```
def __init__(self, attack_type):
    super().__init__()

    self.attack_type = attack_type
    if self.attack_type == AttackType.ROCK:
        self.textures = [
            arcade.load_texture("assets/srock.png"),
            arcade.load_texture("assets/srock-attack.png"),
        ]
    elif self.attack_type == AttackType.PAPER:
        self.textures = [
            arcade.load_texture("assets/spaper.png"),
            arcade.load_texture("assets/spaper-attack.png"),
        ]
    else:
        self.textures = [
            arcade.load_texture("assets/scissors.png"),
            arcade.load_texture("assets/scissors-close.png"),
        ]

    self.scale = self.ATTACK_SCALE
    self.current_texture = 0
    self.set_texture(self.current_texture)
```

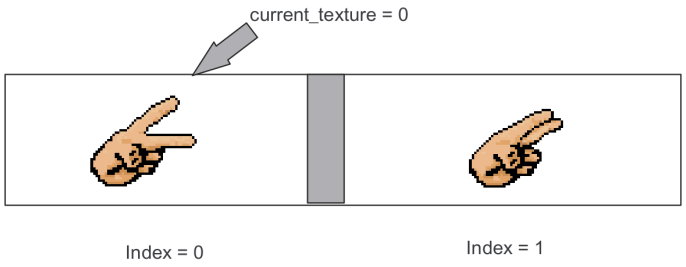
Pour chaque possibilité, on charge en mémoire les deux images nécessaires pour l'animation. La méthode `arcade.load_texture()` retourne un objet de type `Texture` et ce dernier est ajouté automatiquement dans la liste `self.textures`. Ensuite, on ajuste la grosseur de l'image pour qu'elle puisse être encadrée. L'index est initialisé à 0 afin de pointer la première image de la liste et on appelle la méthode `self.set_texture(self.current_texture)` afin que l'image active soit la bonne.

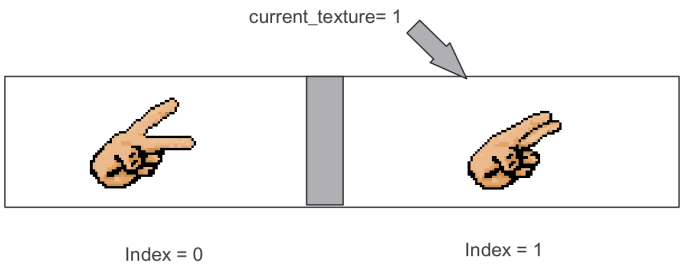
Puisqu'une animation est une séquence d'image, l'attribut `textures` contient la liste de toutes les images de l'animation. L'attribut `texture` (notez le singulier de cet attribut) c'est l'image courante dans la séquence de l'animation. Il faut

aussi une variable qui sert d'index pour l'animation. Vous devez placer ce code dans la classe `AttackAnimation`. Vous allez donc ajouter cette méthode en dessous de la méthode `__init__` de la classe `AttackAnimation`. Voici le code qui permet cette animation:

```
def on_update(self, delta_time: float = 1 / 60):
    # Update the animation.
    self.current_texture += 1
    if self.current_texture < len(self.textures):
        self.set_texture(self.current_texture)
    else:
        self.current_texture = 0
        self.set_texture(self.current_texture)
```

Ce bout de code va afficher la première image de la séquence d'animation dans le "frame" courant. Au "frame" suivant, on incrémente l'index pour afficher l'image suivante de la séquence d'animation. Au "frame" suivant, il n'y a pas de troisième image. On revient donc à l'index 0.





Le va-et-vient entre les deux images fait en sorte que le type d'attaque sera animé.

Avec ce code en place, on peut donc animer les images. Dans la classe principale (MyGame), nous avons besoin d'une animation pour chaque attaque. Ces animations sont des attributs de la classe principale (MyGame). Vous allez donc modifier votre code comme suit:

```
# Il faut remplacer la ligne de code suivante:
self.rock = arcade.Sprite("assets/srock.png")
# Par cette ligne
self.rock = AttackAnimation(AttackType.ROCK)
```

Vous devez le faire pour les deux autres sprites qui représentent les attaques.

Notre `enum` sert donc à paramétriser le "constructeur" de la classe `AttackAnimation`. On utilise alors ces attributs de la même façon que l'on utilisait arcade.Sprite. Il faut invoquer `self.rock.draw()` dans la méthode `on_draw` de notre classe principale pour l'afficher. Il faut aussi invoquer la méthode `self.rock.on_update()` dans la méthode `on_update` de la classe principale (MyGame) afin de faire progresser l'animation.

Avec cela en place, les trois types d'attaques possibles seront animés. Par contre, il y a un problème… L'animation est trop rapide. En effet, avec seulement deux images, 60FPS c'est beaucoup trop. Que devons-nous faire pour remédier à cette problématique? Il faut s'assurer de ne pas faire progresser la séquence d'animation à chaque "frame". Mais comment le faire? Il faut ajouter un accumulateur de temps. Lorsque celui-ci sera atteint, on peut faire progresser la séquence d'animation. Entre-temps, on accumule le temps écoulé entre chaque "frame" dans une variable. Pour ce faire, il faut une constante qui représente la vitesse d'animation. Ensuite, il faut un attribut dans la classe qui servira à accumuler le temps et une qui servira à déterminer combien de temps il faut attendre. Voici ce que ça donne:

```
class AttackAnimation(arcade.Sprite):
    ATTACK_SCALE = 0.50
    ANIMATION_SPEED = 5.0
```

Dans le `__init__`, ajouter le code suivant:

```
self.animation_update_time = 1.0 / AttackAnimation.ANIMATION_SPEED
self.time_since_last_swap = 0.0
```

Ensuite, il faut modifier la méthode `on_update` pour qu'elle contienne notre gestion de temps:

```
def on_update(self, delta_time: float = 1 / 60):
    # Update the animation.
    self.time_since_last_swap += delta_time
    if self.time_since_last_swap > self.animation_update_time:
        self.current_texture += 1
        if self.current_texture < len(self.textures):
            self.set_texture(self.current_texture)
        else:
            self.current_texture = 0
            self.set_texture(self.current_texture)
        self.time_since_last_swap = 0.0
```

TADA! Notre animation sera maintenant fluide!