Jeu vidéo Python

Le but de ce TP est de coder un jeu vidéo d'évitement en Python à l'aide des librairies SDL2 et OpenGL (librairies écrites en C). On utilise "ctypes" pour faire le lien avec le C. OpenGL est une librairie graphique complexe et son utilisation n'est pas nécessaire, mais elle permet de rendre le jeu beaucoup plus fluid

1. Gestion des scores

Commençons par le fichier main.py. C'est ici que se fera la gestion des scores. Un dictionnaire de la forme {'joueur': score} sera enregistré comme une chaîne de caractère dans un fichier texte 'scores.txt'.

Il faut donc vérifier si ce fichier existe. Si ce n'est pas le cas, on le crée en l'ouvrant en mode 'w' et on y écrit '{}' qui représente un dictionnaire vide. On initialise également le dictionnaire. Si le fichier existe déjà, on lit la chaîne contenu et on la transforme en dictionnaire grâce à la fonction ast.literal_eval.

```
def main():
    player_name = ''
    player_score = 0.
    m = menu.Menu()

    if not os.path.exists('scores.txt'):
        with open('scores.txt', 'w') as f:
            f.write('{}')
        scores = {}
    else:
        with open('scores.txt', 'r') as f:
            scores = ast.literal_eval(f.readline())
```

Vient ensuite la boucle Menu -> Jeu. On appelle la fonction main_menu en lui envoyant le nom du joueur pour qu'il n'ait pas à le réécrire à chaque fois, son score et le dictionnaire des scores pour afficher un message s'il a fait un nouveau meilleur score. La fonction renvoie Command.EXIT si le joueur veut quitter. On détermine ensuite le meilleur score en triant le dictionnaire et on lance la partie. On récupère le score de la partie et s'il est meilleur que le meilleur score du joueur, on actualise le dictionnaire et le fichier.

```
while True:
    player_name = main_menu({
        'player_name': player_name,
        'player_score': player_score,
        'scores': scores
    })

    if player_name == Command.EXIT:
        break
    try:
        best_score = list(scores.values())[0]
    except IndexError:
        best_score = 0
    player_score = game.game(best_score)
    if player_score == 0:
        break

    print(player_name, ':', player_score)
    if player_score > scores.get(player_name, 0):
        scores[player_name] = player_score
        scores = dict(sorted(
            scores.items(),
            key=lambda item: item[1],
            reverse=True
        ))
        with open('scores.txt', 'w') as f:
            f.write(scores.__str__())
```

2. Ouverture de la fenêtre

Écrivons le fichier ‘common.py’. Il contient la gestion de la fenêtre et du contexte OpenGL, une classe Settings avec des variables utiles et quelques méthodes.

Pour initialiser la librairie SDL on utilise les fonction suivantes :

```
SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO | SDL_INIT_GAMECONTROLLER)
IMG_Init(IMG_INIT_PNG)
Mix_Init(MIX_INIT_MP3)
TTF_Init()
```

IMG représente SDL_Image, Mix pour SDL_Mixer (musique) et TTF pour SDL_ttf qui permet d’écrire du texte.

Voici la classe Settings :

```
class Settings:
    def __init__(self, initial_width: int, initial_height: int):
        self.music_volume = 10
        self.flags = SDL_WINDOW_RESIZABLE | SDL_WINDOW_OPENGL
        self.fullscreen = True
        self.muted = False
        self.current_w = initial_width
        self.current_h = initial_height
        self.joystick = SDL_NumJoysticks() > 0
        if self.joystick:
            self.joy = SDL_JoystickOpen(0)
            SDL_JoystickEventState(SDL_ENABLE)

    def update_screen(self, w: int, h: int):
        self.current_w = w
        self.current_h = h
        glViewport(0, 0, w, h)

    def toggle_fullscreen(self):
        SDL_SetWindowFullscreen(
                        window,  SDL_WINDOW_FULLSCREEN_DESKTOP  if
self.fullscreen else 0
        )
        self.fullscreen = not self.fullscreen

    def toggle_mute(self):
        if self.muted:
            Mix_Volume(-1, self.music_volume)
        else:
            Mix_Volume(-1, 0)
        self.muted = not self.muted
```

Il y a deux méthodes toggle qui seront appelées par des raccourcis clavier pour couper le son ou mettre en plein écran, et une méthode update_screen utilisée lorsque la taille de la fenêtre a été modifiée. glViewport est une fonction d'OpenGL qui permet de spécifier le rectangle de la fenêtre dans laquelle le rendu graphique est effectué.

```
Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 1024)

screen_info = SDL_DisplayMode()
SDL_GetCurrentDisplayMode(0, screen_info)

settings = Settings(int(screen_info.w / 2), int(screen_info.h / 2))
Mix_Volume(-1, settings.music_volume)
os.environ['SDL_VIDEO_CENTERED'] = '1'
os.environ['SDL_VIDEO_WINDOW_POS'] = '0,0'


window = SDL_CreateWindow(b"Game",
                                                SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,
                          settings.current_w, settings.current_h,
                          settings.flags)

SDL_GL_SetSwapInterval(1)
context = SDL_GL_CreateContext(window)
SDL_GL_MakeCurrent(window, context)
glViewport(0, 0, settings.current_w, settings.current_h)
opengl.init()
```

Mix_OpenAudio initialise l'API Mixer. SDL_GetCurrentDisplayMode permet de récupérer la taille de l'écran afin d'ouvrir une fenêtre de taille initiale (largeur / 2, hauteur / 2).

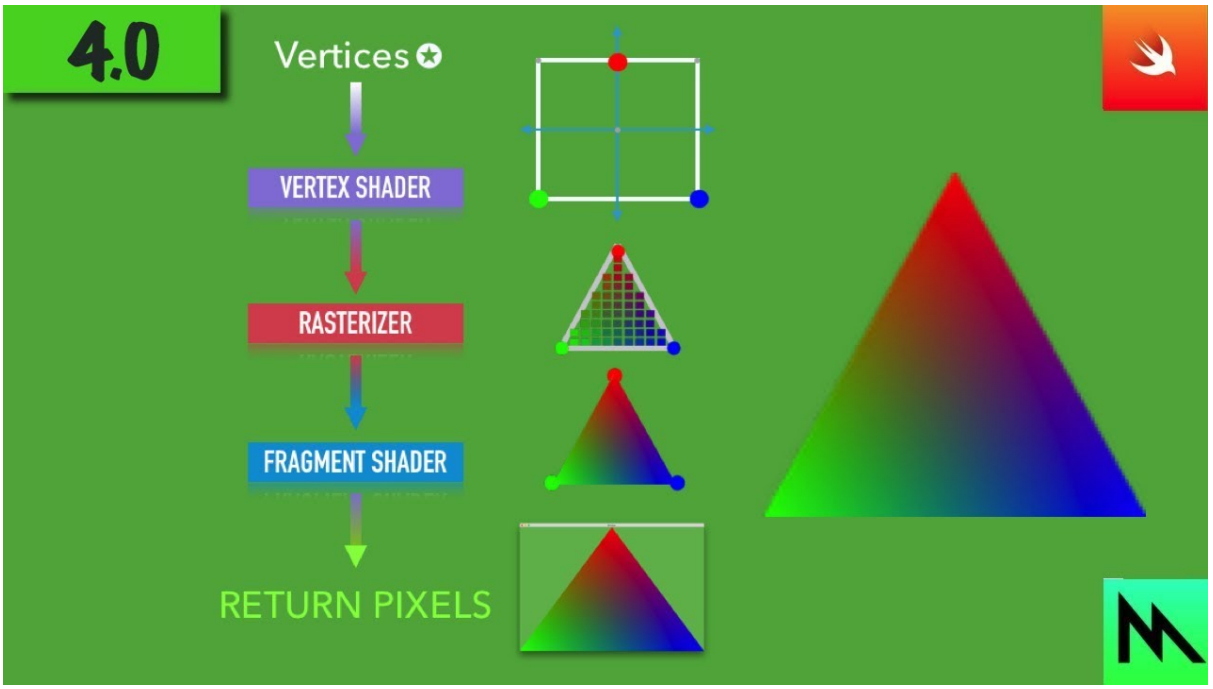
On crée la fenêtre à l'aide de SDL_CreateWindow. La SDL est une librairie écrite en C et nécessite donc des chaînes de caractère similaires, c'est-à-dire un tableau d'octets. On peut aussi utiliser ma méthode .encode(). On utilise les flags SDL_WINDOW_RESIZABLE pour avoir une fenêtre redimensionnable par l'utilisateur et SDL_WINDOW_OPENGL afin de pouvoir gérer le rendu graphique avec OpenGL.

SDL_GL_SetSwapInterval(1) active la synchronisation verticale.

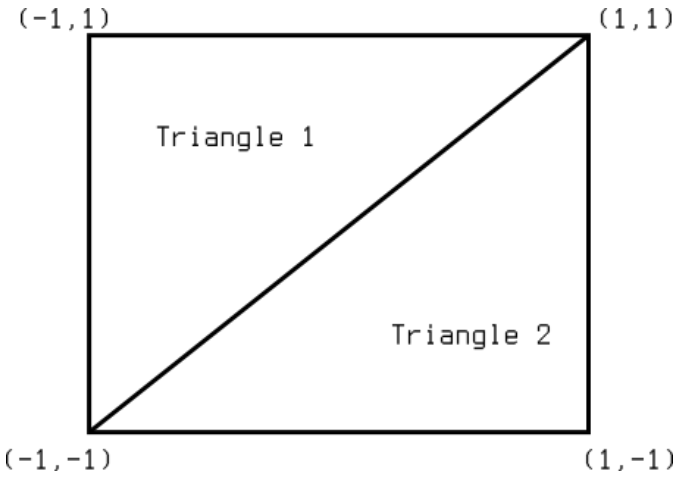
Reste à créer et utiliser le contexte OpenGL.

3. VAO, VBO et shaders

Dans le fichier opengl.py, nous initialisons le VAO (Vertex Array Object) et le VBO (Vertex Buffer Object) nécessaires au bon fonctionnement d'OpenGL (du moins des versions récentes). Le VBO est un buffer de données contenant des vertex (coordonnées de points) et le VAO stocke des informations concernant le VBO. Les shaders sont des bouts de codes écrit en langage GLSL. Ils seront compilés lors de l'exécution du programme et tournent directement sur la carte graphique. Nous utiliserons les plus basiques, le vertex shader et le fragment shader.



OpenGL fonctionne principalement avec des triangles. Ainsi, pour dessiner un carré il faut 2 triangles. Les coordonnées des coins des triangles sont reçues par le vertex shader. La suite est automatique et le fragment shader va ensuite tourner pour chaque pixel du triangle.

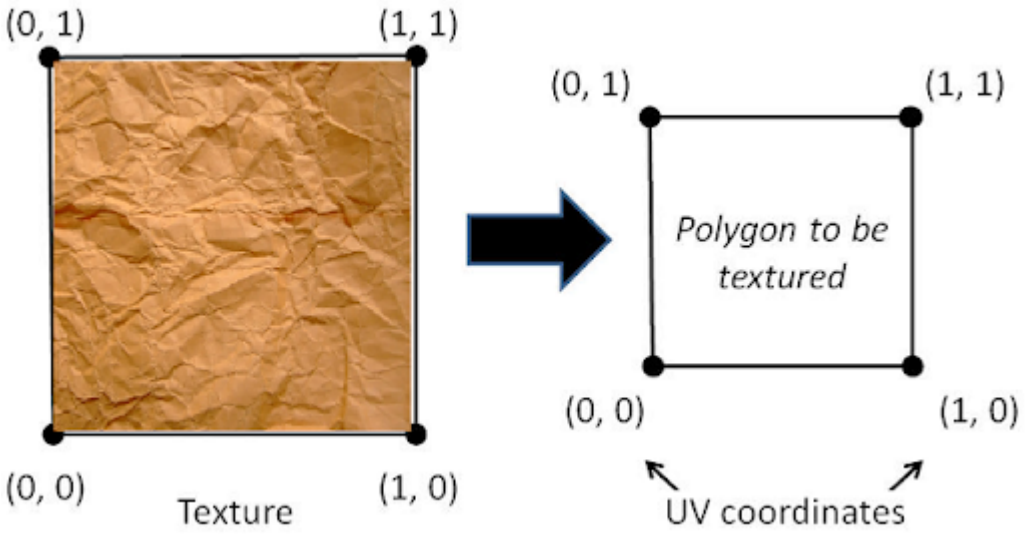


Vertex shader :

Ce shader n'a rien de compliqué dans notre cas mais il est nécessaire.

```
#version 330

layout(location = 0) in vec2 pos;
layout(location = 1) in vec2 uvIn;
out vec2 uv;

void main() {
  gl_Position = vec4(pos, 0, 1);
  uv = uvIn;
}
```

Il reçoit 2 paramètres pour chaque vertex. 'pos' est un vec2, c'est à dire 2 floats. Cette variable représente les coordonnées du point entre -1 et 1 car c'est dans cet intervalle qu'OpenGL travaille et que la valeur de retour du shader doit être. 'uvIn' représente les coordonnées UV de la texture à appliquer dans l'intervalle [0, 1]. Nous afficherons que des rectangles (formés de 2 triangles) avec des png comme texture



gl_Position représente la valeur de sortie du shader, les coordonnées finales. Comme elles seront déjà entre -1 et 1 dans notre jeu, il n'y a rien à changer. Puisque c'est un vec4, il faut rajouter 0 pour la coordonnée z et 1 pour w, ce qui signifie que le vertex est normalisé. La dernière ligne permet simplement de transférer les coordonnées uv au fragment shader puisque qu'on a déclaré au début une variable 'uv' de sortie.

Fragment shader :

```
#version 330

in vec2 uv;
uniform sampler2D tex;
out vec4 fragColor;

void main() {
   fragColor = texture(tex, uv);
}
```

Le fragment shader est exécuté pour chaque pixel dans les triangles décrits par le vertex shader. Son unique but ici est d'appliquer la texture sur nos rectangles.

Il reçoit les coordonnées **interpolées** uv du triangle que nous avons transféré, la texture à utiliser, et renvoie une couleur sous la forme d'un vec4. La fonction 'texture' permet de faire le travail.

Exercices

1. Comment pourrait-on faire pour que tout le rendu graphique soit inversé verticalement ?

Rappel : les coordonnées uv sont dans l'intervalle [0, 1] et (0, 0) est le coin en bas à gauche.

2. Cette fois-ci, essayez de rendre l'image plus 'verte'. La fonction texture renvoie un vec4 au format RGBA. La multiplication de vecteurs se fait composante par composante.

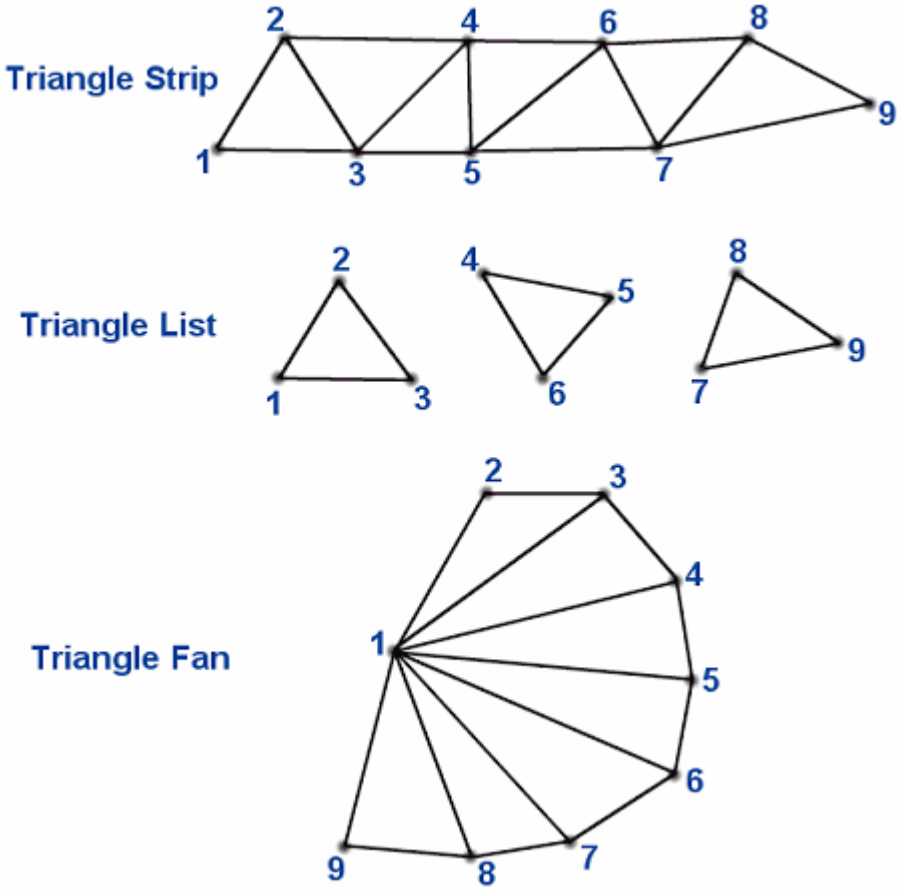
Correction

1. fragColor = texture(tex, vec2(uv.x, 1-uv.y))

2. fragColor = texture(tex, uv) * vec4(0.5, 1, 0.5, 1)

On peut aussi s'amuser avec le canal alpha. Essayez donc de piloter ivre :
fragColor = texture(tex, uv) * vec4(1, 1, 1, 0.3)

Pour dessiner un rectangle, il faut 2 triangles, donc 6 vertex. Or 2 sont communs aux 2 triangles donc 4 suffisent. Pour les réutiliser on va utiliser le flag GL_TRIANGLES_FAN



On aurait tout aussi pu utiliser GL_TRIANGLES_STRIP, mais l'ordre des vertex serait différent.

Il nous faut donc 4*2 float pour le rectangle et de même pour les coordonnées uv donc 16 floats en tout. On prépare un buffer avec des 0 comme suit :

```
vao = glGenVertexArrays(1)
vbo = glGenBuffers(1)

vertex_data = (c_float * 16)(
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0
)

glBindVertexArray(vao)
glBindBuffer(GL_ARRAY_BUFFER, vbo)
glBufferData(GL_ARRAY_BUFFER,  16  *  sizeof(c_float),  vertex_data,
GL_STREAM_DRAW)
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, c_void_p(0))
glEnableVertexAttribArray(0)
glVertexAttribPointer(1,      2,      GL_FLOAT,      GL_FALSE,      0,
c_void_p(8*sizeof(c_float)))
glEnableVertexAttribArray(1)
```

glVertexAtrribPointer enregistre les informations du buffer dans le VAO :

pour la location 0, 2 coordonnées par vertex, de type GL_FLOAT.
pour la location 1, pareil mais le dernier paramètre change, c'est le pointeur vers le début des données. Pour la location 1, c'est la taille de 8 float après la location 0.
Les 8 premiers float concernent donc la position des 4 coins du rectangle et les 8 suivants les 4 coins de la texture à afficher.

Pour afficher une image entière dans toute la fenêtre cela donnerait :

```
vertex_data = (c_float * 16)(
    -1, -1,
    -1, 1,
    1, 1,
    1, -1,

    0, 0,
    0, 1,
    1, 1,
    1, 0
)

glBufferSubData(GL_ARRAY_BUFFER, 0, 16 * sizeof(c_float), vertex_data)
glDrawArrays(GL_TRIANGLE_FAN, 0, 4)
```

4. Lire des images

Avant d'afficher des images il faut déjà pouvoir les lire et les manipuler, c'est le but du fichier images.py.

```
def load(path: str) -> SDL_Surface:
    image = IMG_Load(path)
    image = SDL_ConvertSurfaceFormat(
        image,
        SDL_PIXELFORMAT_RGBA32,
        0).contents
    invert(image)
    return image
```

La fonction load permet d'ouvrir une image à partir de son emplacement. La fonction IMG_Load s'en charge et nous renvoie un pointeur vers une SDL_Surface. On va la convertir au format RGBA32 même si elle l'est probablement déjà car on veut pouvoir manipuler les pixels de l'image. Le ".contents" ici vient de la librairie ctypes et permet de simuler le fonctionnement d'un pointeur en C, il le déréférence et on peut manipuler la SDL_Surface directement. On va devoir inverser l'image verticalement car la SDL et OpenGL ont des axes verticaux opposés, on ne veut pas que les images s'affichent à l'envers.

Voici le format de la structure SDL_Surface :

SDL_Surface

A structure that contains a collection of pixels used in software blitting.

Data Fields

Uint32	**flags**	(internal use)
SDL_PixelFormat*	**format**	the format of the pixels stored in the surface; see SDL_PixelFormat for details (read-only)
int	**w, h**	the width and height in pixels (read-only)
int	**pitch**	the length of a row of pixels in bytes (read-only)
void*	**pixels**	the pointer to the actual pixel data; see Remarks for details (read-write)
void*	**userdata**	an arbitrary pointer you can set (read-write)
int	**locked**	used for surfaces that require locking (internal use)
void*	**lock_data**	used for surfaces that require locking (internal use)
SDL_Rect	**clip_rect**	an SDL_Rect structure used to clip blits to the surface which can be set by SDL_SetClipRect() (read-only)
SDL_BlitMap*	**map**	info for fast blit mapping to other surfaces (internal use)
int	**refcount**	reference count that can be incremented by the application

Les champs qui nous intéressent ici sont w, h, pitch et pixels. On remarque que pitch = w * 4 puisque l'image est au format RGBA32, et a donc 32 / 8 = 4 octets par pixel.

```
def invert(surf: SDL_Surface):
    pitch = surf.pitch
    temp = (c_byte * pitch)()

    for i in range(int(surf.h / 2)):
        row1 = surf.pixels + i * pitch
        row2 = surf.pixels + (surf.h - i - 1) * pitch

        memmove(temp, row1, pitch)
        memmove(row1, row2, pitch)
        memmove(row2, temp, pitch)
```

On va inverser l'image verticalement ligne par ligne, on crée donc une ligne temporaire pour faire l'échange, et on itère sur la moitié des lignes. surf.pixels est un pointeur, mais pas au sens de la librairie ctypes, c'est un entier qui représente une adresse mémoire. L'adresse des lignes à échanger se calcul donc par :

adresse des pixels + i * taille d'une ligne en octet
adresse des pixels + (nombre de lignes - i - 1) * taille d'une ligne en octet

La fonction memmove permet de copier des données en mémoire et donc de faire l'échange en 3 fois.

```
def flip(surf: SDL_Surface):
    pitch = surf.pitch
    temp = (c_byte * 4)()
    for i in range(surf.h):
        for j in range(int(surf.w / 2)):
            p1 = surf.pixels + i * pitch + j * 4
            p2 = surf.pixels + (i+1) * pitch - (j+1) * 4
            memmove(temp, p1, 4)
            memmove(p1, p2, 4)
            memmove(p2, temp, 4)
```

La fonction flip fait la même chose verticalement mais cette fois on doit le faire pixel par pixel car si les octets des lignes se suivent dans la mémoire, ce n'est pas le cas de ceux des colonnes.

5. Le menu

Pour le menu du jeu (menu.py), nous utiliserons une fonction pour chaque sous-menu. Commençons par écrire une fonction générale pour traiter les événements.

```
class Menu:
    class Action(Enum):
        BACK = 1
        ENTER = 2
        QUIT = 3
        UP = 4
        DOWN = 5
        DELETE = 6
        UNICODE = 7

    def __init__(self):
        self.deadzone = 0.7
        self.JOYSTICK_DELAY = 0.130
        self.joy_delay = self.JOYSTICK_DELAY
        self.time = time.monotonic()
        self.unicode = []
```

La classe Action est une énumération qui sert de valeur de retour à la fonction poll_event. Cette dernière traite les événements de la fenêtre et renvoie une liste d'Action qui seront plus simplement traitées par les fonctions des menus.

Voici le sous-menu des meilleurs scores :

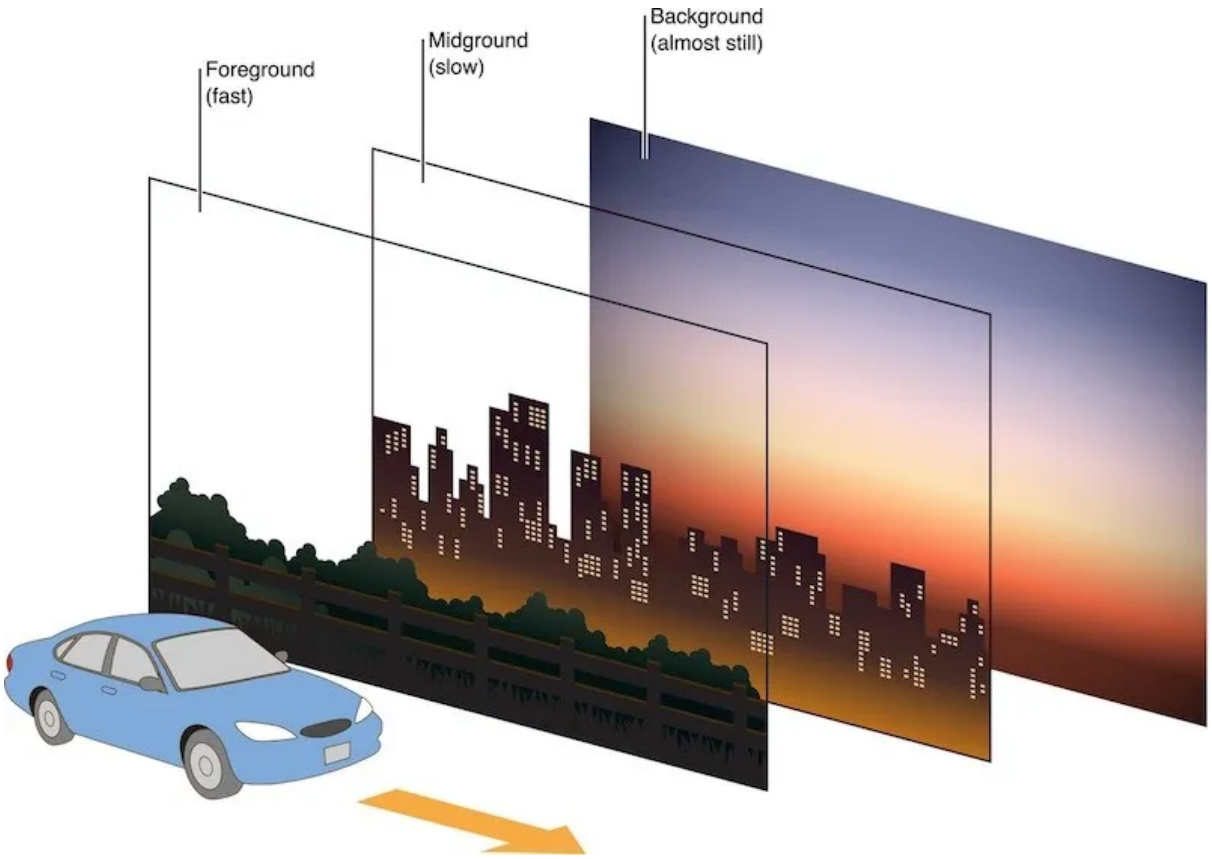
```
def high_scores(self, scores: dict[str, int]):
    n = len(scores)
    while True:
        for event in self.poll_events():
            if event == Menu.Action.QUIT:
                return Command.EXIT
            if event == Menu.Action.BACK:
                return Command.BACK

        glClear(GL_COLOR_BUFFER_BIT)

        for i in range(n):
            text = list(scores.keys())[i] + ' : %.2f' % list(scores.values())[i]
            surf = TTF_RenderUTF8_Blended(font, text.encode(), SDL_Color()).contents
            w = surf.w / settings.current_w * 2
            x = - w / 2
            h = surf.h / settings.current_h * 2
            y = h + (20 + surf.h) * (n / 2 - i) / settings.current_h * 2
            blit(x, y, w, h, surf)

        SDL_GL_SwapWindow(window)
        SDL_Delay(20)
```

On reçoit le dictionnaire des scores triés. On actualise la fenêtre toutes les 20 millisecondes. Si le joueur quitte le jeu, on renvoie Commande.EXIT. S'il utilise la touche 'Retour' on renvoie Command.BACK.

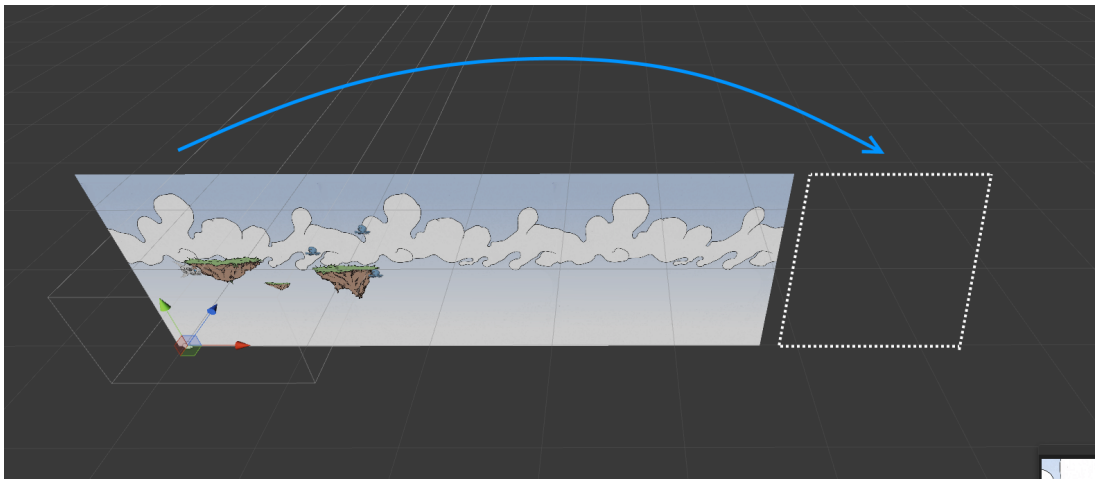
On clear la fenêtre (avec du noir) et on affiche les noms et scores à deux décimales près des joeurs avec une booucle.

6. Fond du jeu : Parallax

Pour le fond du jeu, plutôt que d'utiliser simplement une image, nous allons utiliser un parallax. Cela consiste en plusieurs images qui se superposent et défilent à des allures différentes :



Les images sont prévues pour être répétées, ce qui donne une impression de défilement infini. Lorsqu'on arrive au but d'une d'entre elle il faut donc afficher une seconde fois la même image :



OpenGL nous simplifie la tâche. On peut spécifier le comportement du rendu lorsque les coordonnées uv sortent de l'intervalle [0,1] grâce au Texture Wrapping :

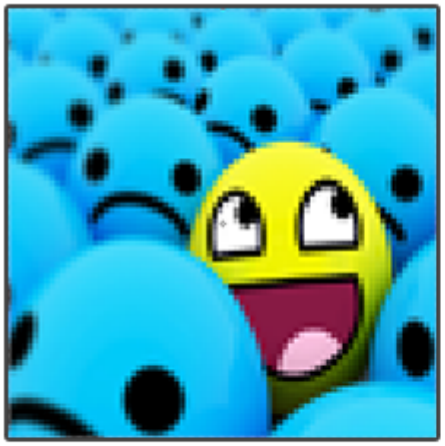
GL_REPEAT GL_MIRRORED_REPEAT GL_CLAMP_TO_EDGE GL_CLAMP_TO_BORDER

L'option qui nous intéresse est GL_REPEAT. Grâce à elle, on n'a pas à se préoccuper de la répétition du fond.

Commençons d'abord par écrire une classe Texture (texture.py).

```
class Texture:
    def __init__(self, path: str, flipped: bool = False):
        surf = image.load(path)
        if flipped:
            image.flip(surf)
        self.id = glGenTextures(1)
        glBindTexture(GL_TEXTURE_2D, self.id)
        glTexImage2D(
            GL_TEXTURE_2D, 0,
            GL_RGBA,
            surf.w, surf.h, 0,
            GL_RGBA, GL_UNSIGNED_BYTE,
            c_void_p(surf.pixels)
        )
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER)
        glBindTexture(GL_TEXTURE_2D, 0)
        self.image = surf
        self.w = surf.w
        self.h = surf.h

    def resize(self, w: int, h: int):
        self.w = w
        self.h = h
```

On commence par charger l'image et la retourner verticalement si voulu. On génère une texture OpenGL et on y met notre SDL_Surface en précisant un pointeur ayant pour adresse surf.pixels. Reste à spécifier deux paramètres de la texture, le Texture Wrapping expliqué précédemment ainsi que le Texture Filtering qui détermine le comportement de la texture lorsque la surface n'est pas aux dimensions exacte de l'image. :



GL_NEAREST

GL_LINEAR

Pour la parallax (parallax.py), nous allons utiliser 2 classes. La classe Layer représente une image de fond défilant à une vitesse donnée, et la classe Parallax représente un ensemble de Layer. On pourra hériter de cette classe pour préciser les images à utiliser, les vitesses de défilement et les positions verticales des images. Voici la classe Layer :

```
class Layer:
    def __init__(self, tex: Texture, speed: float, y1: float, y2:
float):
        self.w = tex.w
        self.h = tex.h
        self.y1 = y1
        self.y2 = y2
        self.speed = speed
        self.scrolling = 0.
        self.tex = tex
        glBindTexture(GL_TEXTURE_2D, tex.id)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
        glBindTexture(GL_TEXTURE_2D, 0)
```

On y précise la texture utilisée, la vitesse et les 2 hauteurs relativement au haut et au bas de la fenêtre. On change également le Texture Wrapping à GL_REPEAT. La texture OpenGL affectée par les fonctions est celle qui est actuellement 'bind'. Il faut donc toujours 'bind' une texture avant de pouvoir l'utiliser.

La méthode update est appelée à chaque frame

```
def update(self, delta: float):
    self.scrolling += self.speed * delta / 1000 * settings.current_w
                    / self.w / settings.current_h * self.h
    if self.scrolling >= 1:
        self.scrolling = 0
    self.render()
```

La variable ‘scrolling’ est un float entre 0 et 1 qui représente le défilement de l’image. L’image se décale vers la gauche et elle sort de l’écran lorsque scrolling = 1.

Le méthode update se charge de mettre à jour cette variable en respectant les dimensions de l’écran, la vitesse du Layer et la variable ‘delta’. A chaque frame du jeu, delta représente le temps passé durant la frame en millisecondes et est utilisé pour toutes les update.

Grâce à delta, le fond défilera à la même vitesse quelque soit les FPS du jeu. De la même façon, il sera stable par rapport aux dimensions de l’écran puisqu’on en tient compte.

Il en sera de même pour les déplacements de l’avion et tout le reste...

Exercices

1. Comment peut-on inverser la direction du défilement ?

2. Essayez de commenter la ligne glTexParameteri, ce qui signifie à changer GL_REPEAT en GL_CLAMP_TO_BORDER car on l’a spécifié comme comportement par défaut dans la classe Texture. Vous verrez mieux comment fonctionne le parallax.

Correction

1. Il faut modifier le “scrolling += “ en “scrolling -= “
   et le “if scrolling >= 1: scrolling = 0” en “if scrolling <= 0: scrolling = 1”

```
def render(self):
    window_w = settings.current_w * self.h / settings.current_h / self.w
    vertex_data = (c_float * 16)(
        -1, self.y2,
        1, self.y2,
        1, self.y1,
        -1, self.y1,
        self.scrolling, 0,
        self.scrolling + window_w, 0,
        self.scrolling + window_w, 1,
        self.scrolling, 1
    )
    glBufferSubData(GL_ARRAY_BUFFER, 0, 16 * sizeof(c_float), vertex_data)
    glBindTexture(GL_TEXTURE_2D, self.tex.id)
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4)
    glBindTexture(GL_TEXTURE_2D, 0)
```

Dans la méthode render, on calcule la largeur de la fenêtre mise à l’échelle pour avoir la même hauteur que le parallax.*

On multiplie la largeur de l’écran par le ratio *image_h / screen_h* et on divise par la largeur de l’image pour être entre 0 et 1.

On crée ensuite le buffer pour les vertex, d'abord les 4 coins pour la position puis pour les textures. On utilise cette fois-ci glBufferSubData avec une offset de 0 au lieu de glBufferData car cette fonction permet à OpenGL de ne pas ré-allouer de la mémoire pour les buffer mais de réutiliser l’espace précédent, l’allocation étant une opération lente.

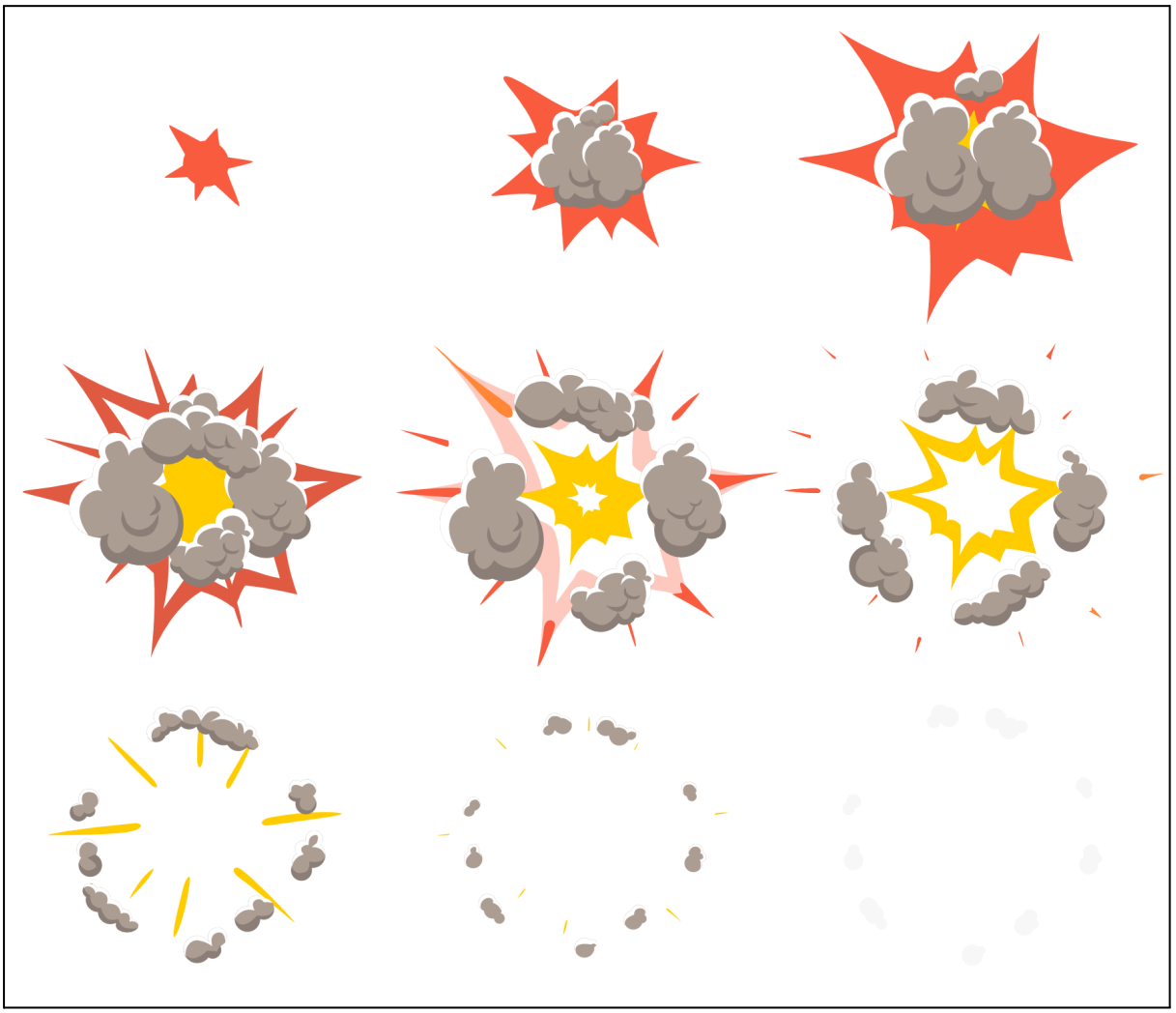
Dans la classe Parallax, on récupère le chemin vers les images à partir du nom de la classe qui hérite de Parallax. On utilise une liste de Layer, et on récupère les informations ‘y1’, ‘y2’ et ‘speed’ grâce à des fonctions qu’il faudra redéfinir pour chaque sous-classe.

Pour que l’avion et les oiseaux passent derrière les nuages, il faut actualiser les Layer en 2 fois. La variable ‘first_half’ est un tuple avec les index des Layer à actualiser en premier.

```
self.layers: list[Layer] = []
file_names = sorted(
    glob.glob(file_path(os.path.join(self.path, 'parallax', '*.png')))
)

for i in range(len(file_names)):
   y1 = self.gen_height_from_top(i)
   y2 = self.gen_height_from_bottom(i)
   speed = self.gen_speed(i)
   tex = Texture(file_names[i])
   tex.resize(int(tex.w * (y1 - y2) / 2), tex.h)
   layer = Layer(tex, speed / 8, y1, y2)
   layer.scrolling = 0
   self.layers.append(layer)

self.first_half = tuple(range(len(self.layers)))
```

7. Animations

Pour créer des animations en jeu, nous allons utiliser des images comme celle-ci :



En affichant une partie de l'image à la fois, l'explosion donne l'impression d'être animée. On a besoin de connaître le nombre d'images horizontalement, verticalement, et en tout. Les coordonnées uv feront le reste.

Voici la classe Animation :

```
class Animation:
    def __init__(self,
                 tex: Texture,
                 sprites: int,
                 format: Vec2,
                 ratio: Vec2,
                 duration: int,
                 mask_needed: bool):
        self.tex = tex
        self.sprites = sprites
        self.format = format
        self.ratio = ratio
        self.w = 2 * self.ratio.x
        self.h = 2 * self.ratio.y
        self.w_tex = 1 / self.format.x
        self.h_tex = 1 / self.format.y
        self.duration = duration
        if mask_needed:
            self.mask = mask.Mask(self)
        self.resize()
```

Les paramètres sont :

- tex : La texture associée
- sprites : Le nombre d'images en tout
- format : Le nombre d'images horizontalement et verticalement
- ratio : La taille que devra avoir l'image par rapport à la fenêtre (entre 0 et 1)
- duration : La durée de l'animation en millisecondes
- mask_needed : Vrai si l'animation servira pour les collisions

On multiplie par 2 self.ratio.x car l'intervalle [-1, 1] représentant la fenêtre OpenGL est de mesure 2. On inverse self.format pour avoir la largeur et la hauteur pour la texture. S'il y a deux images horizontalement par exemple, une sous-image sera de largeur ½.

La méthode render :

```
def render(self, x: float, y: float, index: int):
    x_tex = (index % self.format.x) / self.format.x
    y_tex = (self.format.y - 1 - (index // self.format.x)) / self.format.y

    vertex_data = (ctypes.c_float * 16)(
        x, y,
        x + w, y,
        x + w, y + h,
        x, y + h,

        x_tex, y_tex,
        x_tex + w_tex, y_tex,
        x_tex + w_tex, y_tex + h_tex,
        x_tex, y_tex + h_tex
    )
    glBufferSubData(GL_ARRAY_BUFFER, 0, 16 * sizeof(c_float), vertex_data)
    glBindTexture(GL_TEXTURE_2D, self.tex.id)
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4)
    glBindTexture(GL_TEXTURE_2D, 0)
```

Ici, on calcule les coordonnées uv du coin en bas à gauche sur la texture. On utilise pour ça le quotient et le reste de la division euclidienne de l'index (le numéro de l'image) par la largeur.

Si l'image est au format (3, 3) comme l'explosion montrée au-dessus, l'index de l'image au milieu à droite est 5. On calcule les coordonnées (2, 1) en faisant :

- 5 % 3 = 2 : index % format.x
- 5 // 3 = 1 : index // format.x

Car 5 = 1 x 3 + 2.

Il faut diviser les coordonnées par format.x et format.y pour être entre 0 et 1. Cela s'arrêterait là si les axes d'OpenGL étaient les mêmes que ceux de l'image, mais comme celle-ci se lit de haut en bas il faut inverser la coordonnée y en faisant format.y - 1 - y. Ainsi, la ligne 0 se transforme en 3 - 1 - 0 = 2 et la ligne 2 en 3 - 1 - 2 = 0.

On a donc notre résultat : (2, 1) qui est le même car la case est sur la ligne centrale. Reste à diviser : (⅔, ⅓).

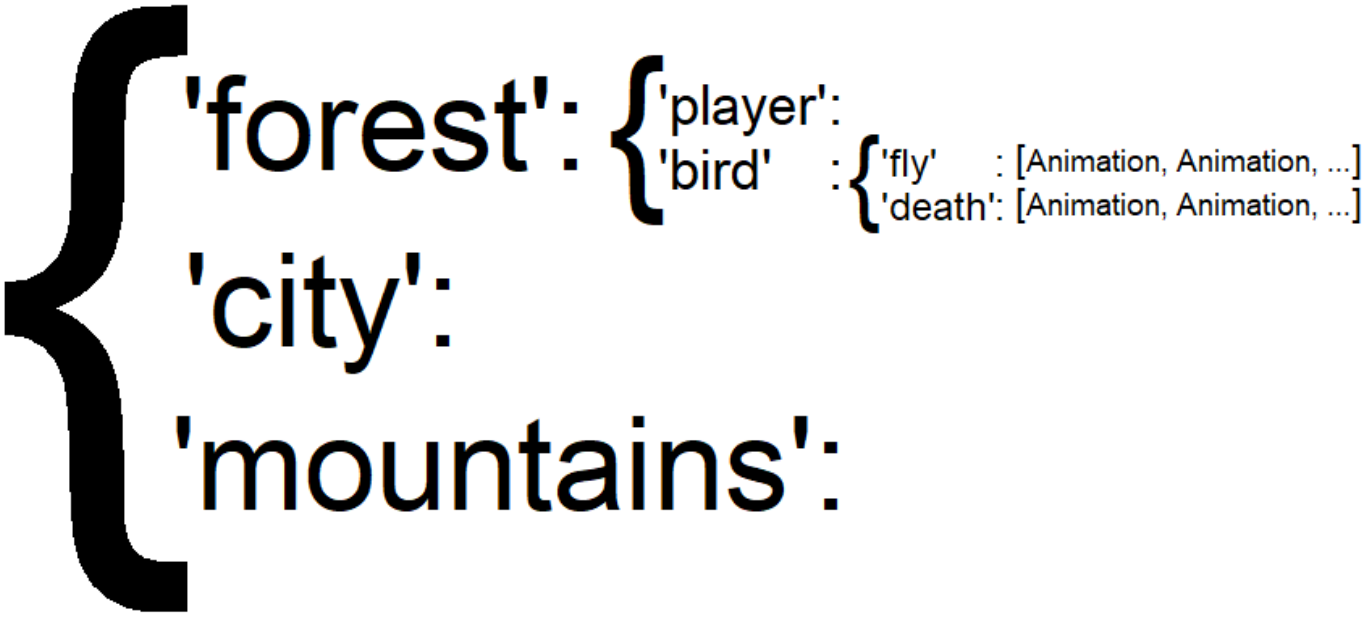
8. Ressources

Au début du jeu, on va initialiser les animations utilisées. C'est fait dans le fichier ressources.py. Nous allons utiliser un dictionnaire par niveau et les ranger dans un dictionnaire global : dictionnary.

Pour chaque niveau, il y a d'autres dictionnaires qui représentent les entités du jeu. Elles possèdent parfois plusieurs animations, comme l'oiseau qui en a une lorsqu'il vole et une autre lorsqu'il est touché et meurt.

Pour chaque entité, ce dictionnaire a des actions comme 'fly' ou 'death' rangées dans d'autres dictionnaires. Finalement, ces derniers contiennent des listes d'animations. Il peut y en avoir plusieurs, par exemple pour les avions de différentes couleurs. Une est choisie au hasard.

Voici un schéma :



On utilise une fonction 'load' pour charger les images, créer les Textures et les Animations associées, puis, les ranger dans le dictionnaire.

```
def load(path: str,
         ratio: Vec2,
         sprites: int = 1,
         sprite_format: Vec2 = Vec2(1, 1),
         duration: int = 1000,
         mask: bool = False,
         flipped: bool = False):

    file_names = glob.glob(file_path(path))
    try:
        level, _, entity, action, _ = path.split('/')
    except ValueError:
        level, _, entity, _ = path.split('/')
        action = 'fly'
    if flipped:
        action += '_flipped'
    if level not in dictionary:
        dictionary[level] = {}
    if entity not in dictionary[level]:
        dictionary[level][entity] = {}
    if action not in dictionary[level][entity]:
        dictionary[level][entity][action] = []
    for i in range(len(file_names)):
        tex = Texture(file_names[i], flipped)
        anim = Animation(tex, sprites, sprite_format, ratio, duration, mask)
        dictionary[level][entity][action].append(anim)
```

On découpe le chemin des fichiers pour connaître le nom du niveau, de l'entité et de l'action. S'il n'y a pas de dossier pour l'action, on suppose que c'est 'fly'. On crée des Textures et des Animations à partir des images. Voici un exemple de l'utilisation de cette fonction pour la mort des oiseaux :

```
load('mountains/sprites/bird/death/*.png', Vec2(0.04, 0.05), 5,
Vec2(5, 1), duration=600, flipped=True)
```

Le premier Vec2 sert à préciser la taille de l'animation. On précise qu'il y a 5 images par animation. 5 horizontalement et une seule verticalement. La durée est de 600 millisecondes et on veut la charger inversée horizontalement. Elle n'est pas utilisée pour les collisions.

Pour le coeur indiquant le nombre de vie :

```
load('global/sprites/hp/*.png', Vec2(0.02, 0.03))
```

On ne précise que la taille. Il n'y a alors par défaut qu'une seule image dans l'animation.

9. Entités

Dans cette partie, nous allons nous occuper de créer des classes pour les entités telles que l'avion ou les oiseaux. Ces classes vont hériter de la classe Entity (entity.py).

```
class Entity:
    def __init__(self,
                 animations: dict[str: list[Animation]],
                 pos: Vec2,
                 vel: Vec2,
                 flipped: bool = False):
        self.animations = animations
        anim = random.choice(animations[('fly', 'fly_flipped')[flipped]])
        self.vel = vel
        self.rect = SDL_Rect(0, 0, 0, 0)
        self.pos = Vec2(
            pos.x - (anim.ratio.x * (pos.x + 1)),
            pos.y - (anim.ratio.y * (pos.y + 1))
        )
        if pos.x < -1:
            self.pos.x = -1 - anim.ratio.x * 2
        elif pos.x > 1:
            self.pos.x = 1
        if pos.y < -1:
            self.pos.y = -1 - anim.ratio.x * 2
        elif pos.y > 1:
            self.pos.y = 1
        self.time = random.random() * anim.duration
        self.index = 0
        self.anim = anim
        if hasattr(anim, 'mask'):
            self.mask = anim.mask.data[self.index]
        self.resize()
        self.dead = False
        self.vy_time = 0
```

Les variables 'pos' et 'vel' représentent la position et la vitesse de l'objet. On envoie aussi un dictionnaire de listes d'animations provenant du fichier ressources. Le booléen 'flipped' précise si l'image doit être retournée horizontalement.

La variable 'rect' est un SDL_Rect, c'est à dire 4 entiers. 2 pour la position et 2 pour la largeur et la hauteur. Comme c'est des entiers, on peut les utiliser pour calculer les collisions.

Avec un petit calcul, on fait en sorte que l'entité soit collée à gauche si pos.x = -1 et collée à droite si pos.x = 1. De même verticalement. On vérifie également que pos est entre -1 et 1. Si ce n'est pas le cas pour un des axes, on colle l'entité au bord de la fenêtre mais du côté extérieur. C'est utile pour que les ennemis n'apparaissent pas d'un coup dans la fenêtre.

La variable 'vy_time' sert à ajouter une variation de la vitesse verticale pour rendre les déplacements plus réalistes.

La classe possède une méthode 'update' qui met à jour la position en fonction de la vitesse et appelle la méthode 'render' de son animation. Elle renvoie True si l'entité est morte et que son animation de mort est terminée.

Voici la méthode 'collide' :

```
def collide(self, other: Entity) -> bool:
    return hasattr(other, 'mask') and hasattr(self, 'mask') and not (
            (self.rect.x >= other.rect.x + other.rect.w)
            or (self.rect.x + self.rect.w <= other.rect.x)
            or (self.rect.y >= other.rect.y + other.rect.h)
            or (self.rect.y + self.rect.h <= other.rect.y)) \
           and mask.collide(self.mask, other.mask, self.rect, other.rect)
```

Elle teste si les 2 entités ont un attribut 'mask', si les rectangles associées sont en collisions, puis si les masques se touchent. Pour simplifier on peut enlever la dernière ligne et simplement utiliser les collisions des rectangles qui sont faites avec la méthode AABB (axis aligned bounding box).

10. Mouvements du joueur

On peut à présent hériter de la classe Entity pour créer la classe Player (player.py)

```
class Player(Entity):
    deadzone = 0.25
    friction = 0.7

    def __init__(self, level: str):
        Entity.__init__(self,
                ressources.get(level, 'player'),
                Vec2(0, 0), Vec2(0, 0))

        self.hp = 3
        self.max_speed = 100
```

La deadzone est utilisée pour la manette. Cela permet au joueur de ne pas se déplacer s'il ne touche pas son joystick. Les joysticks reviennent au centre mais pas parfaitement, ainsi avec une deadzone de 0.25, ses mouvements ne seront pas comptés s'il n'est pas poussé à plus de 25%.

La friction est un coefficient permettant à la vitesse du joueur de diminuer d'elle-même avec le temps. Elle rend les déplacements plus réalistes et plus fluides.

Dans la méthode update, on va d'abord gérer les entrées clavier/manette, puis appeler la méthode Entity.update, et finalement restreindre les déplacements à l'intérieur de la fenêtre.

Lorsque le joueur utilise les flèches pour aller en diagonale, on fait attention à multiplier le rapport de vitesse par √2 / 2 (0.707) afin que l'avion n'aille pas plus vite qu'en ligne droite :

```
if keys[SDL_SCANCODE_UP] ^ keys[SDL_SCANCODE_DOWN]:
    self.vel.y += (delta, -delta)[keys[SDL_SCANCODE_DOWN]] * \
                  (1, 0.707)[keys[SDL_SCANCODE_LEFT] ^ keys[SDL_SCANCODE_RIGHT]]

if keys[SDL_SCANCODE_LEFT] ^ keys[SDL_SCANCODE_RIGHT]:
    self.vel.x += (delta, -delta)[keys[SDL_SCANCODE_LEFT]] * \
                  (1, 0.707)[keys[SDL_SCANCODE_UP] ^ keys[SDL_SCANCODE_DOWN]]
```

11. Les niveaux

Tout comme la classe Parallax et la classe Entity, la classe Level (level.py) est faite pour être héritée. On y écrit du code général.

Dans le constructeur on récupère le nom du niveau et on l'utilise pour récupérer la classe parallax associée, pour trouver les musiques du niveau et charger l'avion du niveau.

Chaque niveau possède un avion et une liste d'entités : 'self.flying'

La méthode update renvoie True lorsque le niveau est terminé, c'est-à-dire quand l'avion est détruit.

```
def update(self, delta, keys, joy_value) -> bool:
    self.background.update1(delta)
    if self.plane.update(delta, keys, joy_value):
        return True
    for f in self.flying:
        if f.update(delta):
            self.flying.remove(f)
    self.background.update2(delta)
    self.hp_bar.update(delta, self.plane.hp)

    SDL_GL_SwapWindow(window)

    for f in self.flying:
        if self.plane.collide(f):
            self.plane.hp -= 1
            if self.plane.hp == 0:
                self.plane.die()
                self.plane.copy_vel(f)
            f.die()

    return False
```

On actualise la première moitié du fond, puis l'avion avec les variable 'keys' et 'joy_value' pour le clavier et les manettes, et finalement les entités de la liste 'flying'. On actualise ensuite le reste du parallax et les coeurs pour les points de vie.

Il ne reste plus qu'à tester les collisions entre l'avion et les autres entités.

Pour la suite, nous allons avoir besoin d'une classe Timer (timer.py) afin de faire apparaître des ennemis régulièrement au fil du temps. Voici son code :

```
class Timer:
    def __init__(self, delay: float):
        self.delay = float(delay)
        self.delta = 0

    def update(self, delta):
        self.delta += delta
        n = int(self.delta / self.delay)
        self.delta -= self.delay * n
        return n
```

Cette classe est très simple. Elle possède un delay qui est la fréquence de l'événement associé. A chaque appel de la méthode update, elle cumule le delta et renvoie le nombre de fois que 'delay' millisecondes sont passées depuis la dernière fois. Ainsi même si on ne l'appelle très souvent et que le delay est très petit, on ne ratera aucun événement.

On peut maintenant écrire le niveau des montagnes (levels.mountains.py)

```
class Mountains(Level):
    def __init__(self):
        Level.__init__(self)
        self.add_bird_timer = Timer(1000)
        self.add_bunny_timer = Timer(5000)

    def update(self, delta, *args) -> bool:
        for i in range(self.add_bunny_timer.update(delta)):
            self.flying.append(
                Bunny(Vec2(rand(), 1.1), Vec2(-20, -30), True)
            )
        for i in range(self.add_bird_timer.update(delta)):
            speed = -20 + rand() * 6 - 3
            flipped = False
            if rand() < 0.5:
                flipped = True
                speed = -170 + rand() * 20 - 10
            self.flying.append(
                Bird(Vec2(1.1, rand() * 2 - 1), Vec2(speed, 0), flipped)
            )
        return Level.update(self, delta, *args)
```

On fait simple : un timer pour les lapins et un pour les oiseaux. Il y aura donc un oiseau toutes les secondes et un lapin toutes les 5 secondes.

On utilise une boucle for pour chaque timer. Ainsi, si le timer des oiseaux renvoie 2, c'est qu'entre 2 et 3 secondes sont passées depuis le dernier appel.

Pour les lapins, on lui donne une position horizontale entre 0 et 1 avec la fonction rand(). Il sera donc dans la moitié droite de l'écran. Comme il tombe verticalement, on lui donne une position verticale supérieure à 1. On a précisé dans le code de Entity que dans ce cas on colle l'entité au bord extérieur de la fenêtre. Sa vitesse (-20, -30) le fait se déplacer vers le coin en bas à gauche.

Pour les oiseaux il y a deux cas. Soit un oiseau est tourné dans le même sens que l'avion, mais va moins vite et donc se déplace lentement de droite à gauche, soit il est à contre-sens et va donc très vite.

Exercices

1. Comment faire pour que les oiseaux apparaissent de plus en plus vite ?

2. Peut-on faire en sorte que seul l'avion passe derrière les nuages mais pas les oiseaux ni les lapins ?

Correction

1. On peut simplement ajouter la ligne suivante au début de la boucle ‘for’ du timer des oiseaux :

```
for i in range(self.add_bird_timer.update(delta)):
  self.add_bird_timer.delay -= (self.add_bird_timer.delay - 400) / 50
  ...
```

Ou alors on peut déclarer un autre timer avec un délai d’une seconde par exemple l’utiliser pour diminuer la fréquence du timer des oiseaux de quelques pourcents.

```
self.more_birds_timer = Timer(1000)
...
...
for i in range(self.more_birds_timer.update(delta)):
   self.add_bird_timer.delay *= 0.95
```

2. Il suffit d’inverser quelques lignes dans le code de la classe Level :

```
def update(self, delta, keys, joy_value) -> bool:
   self.background.update1(delta)
   if self.plane.update(delta, keys, joy_value):
       return True
   self.background.update2(delta)
   for f in self.flying:
       if f.update(delta):
           self.flying.remove(f)
   ~~self.background.update2(delta)~~
   self.hp_bar.update(delta, self.plane.hp)

   SDL_GL_SwapWindow(window)

   ...
```

12. Lancer le jeu

Dans cette partie on s'occupe du fichier game.py. La fonction 'game' reçoit le meilleur score enregistré et doit lancer le jeu. Lorsque le joueur bat le meilleur score, un son audio est joué. La fonction renvoie le temps du joueur.

```
def game(best: int) -> float:
    global start_time
    start_time = time.time()
    ressources.resize()
    hp = 3
    for level in (Mountains(), City(), Forest()):
        command = play(level, best, hp)
        if command == Command.EXIT:
            return 0
        if command == Command.BACK:
            break
        if command == Command.NEXT:
            print('Tricheur !')
        hp = level.plane.hp

    return time.time() - start_time
```

Dans la fonction play, on va surtout gérer les évènements de la fenêtre, du clavier et des manettes. On utilise un 'delta' fixe plutôt que celui qu'on calcule car il n'est pas très précis et le jeu devient moins fluide. On peut le faire car ici les FPS sont bloqués à 60 par la SDL. S'ils descendent en dessous en revanche, le jeu ralentit.