

**Groupe:**

**Dana A.**

**Ines S.**

**Soléna Dubus**

**Terminale S**

**Spécialité ISN**



UPER DINO









**0- SOMMAIRE**

**I- Notre projet page 3**

**II- Répartition des tâches et démarche collaborative page 5**

**III- Réalisation personnelle page 6**

**IV – Problèmes, erreurs et résolutions page 17**

**V- Bilan et perspectives page 18**

**VI – Annexe page 18**

**I- NOTRE PROJET**

1. **Introduction**

Pour notre projet, nous avons décidé de créer un jeu vidéo de plateforme sous forme d’un « pixel game » (jeu formé de pixels de grandes tailles, « visibles »). En s’inspirant notamment des pères fondateurs du jeu vidéo, nous voulions produire un jeu similaire à Mario Bros. Ainsi donc, notre jeu serait donc visible en 2 dimensions et permettrait au personnage de sauter, d’avancer, reculer et d’interagir avec son environnement (collision, etc…)

1. **Notre scénario**

Un beau jour, des archéologues en plein milieu de leur travail découvrirent un œuf gigantesque, et fossilisé. Plein de curiosité, ils décidèrent de le confier à un ensemble de chercheurs de la compagnie « LEDA ». Durant un des tests, les scientifiques réussirent à redonner vie à l’être au sein de l’œuf, qui s’avérait être un dinosaure. Par l’ensemble de ces tests, le dinosaure était devenu intelligent (et… radioactif). Son but, dès sa naissance, était de retrouver sa liberté, et donc de revenir à l’ère Jurassique afin de retrouver sa famille. Pour cela, le dinosaure entreprit de s’échapper du laboratoire et de prendre une machine à remonter dans le temps (créé par le laboratoire). C’est ainsi que commença l’aventure de… SUPER DINO !

1. **Pourquoi ce projet ? Pourquoi un jeu vidéo ?**

En choisissant la spécialité ISN, une partie du groupe avait d’ores et déjà envie de créer un jeu vidéo (de tout type possible). On trouvait que c’était un projet à la fois créatif et ambitieux (c’est pour cela que l’on trouvait l’idée intéressante). Qui plus est, le jeu vidéo est un domaine qui nous intrigue beaucoup car il est fréquent de se demander lorsque l’on joue chez nous comment un jeu peut-il être fait et conçu.

1. **But et objectifs, et découpage du jeu**

Premièrement, notre but est de créer un jeu vidéo de A à Z à l’aide de Python (un langage informatique) et Pygame (bibliothèque d’extensions de Python). Plus précisément, nous avions divisé notre projet en de nombreuses étapes, même si nous nous sommes rendu compte au fur et à mesure que la tâche était trop importante.

Ainsi donc, notre but originel était de produire une interface d’accueil, proposant l’option de jouer suite à une commande. Ensuite, nous voulions de cette interface accéder à notre jeu vidéo, composé donc d’une plateforme au sein duquel un personnage pourrait se déplacer et sauter selon les intentions du joueur. Si le joueur tombait, il mourrait et tombait sur une image « Game Over ». S’il finit le niveau, il tombe sur un écran « Bravo ! »

Afin de rendre notre code plus lisible, nous avons voulu diviser le programme avec l’utilisation d’un code principal (squelette du code) qui aurait recours à un autre code « source » qui contiendrait l’ensemble des informations importantes (informations sur les personnages, le décor du niveau, les différents blocs de plateformes, etc…)

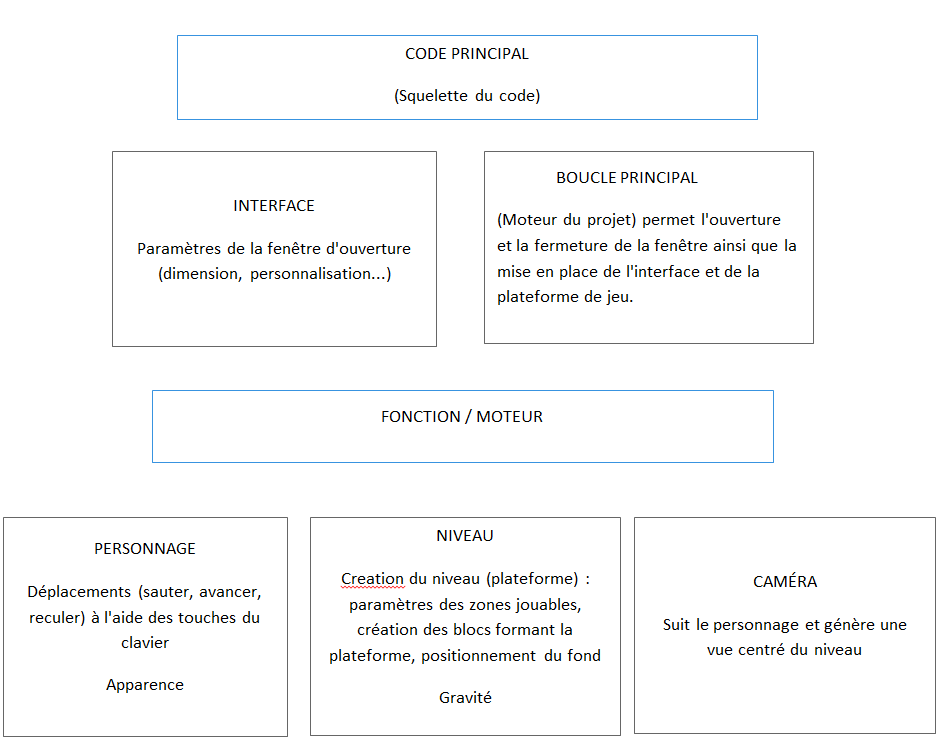
Cependant, au final, nous nous sommes heurtés à un problème : notre premier code ne marchait pas vraiment, nous n’arrivions pas à intégrer une gravité et une caméra suivant le joueur. Nous sommes donc allées chercher les réponses sur internet et sommes tombées sur un code « moteur » qui regroupait tous ce dont l’on avait besoin. Ainsi, en s’inspirant de ce code, nous avons obtenu une seconde version de notre jeu, qui intègre pratiquement tout ce que l’on souhaitait, mais qui se retrouve à être très différent de notre premier. Afin de présenter l’ensemble de notre travail, nous avons donc décidé de montrer et expliquer notre première version du jeu, puis ensuite la seconde. Les deux jeux se divisent de la façon suivante :

|  |  |
| --- | --- |
| **SUPER DINO 1 (incomplet)** | **SUPER DINO 2** |
| I- Interface menu du jeu   1. Traitement de l’image sur pygame 2. Chargement des fonctions et classes du jeu 3. Notion d’évènements   II- Le jeu   1. Le fond et les plateformes (construction du niveau) 2. Le Joueur 3. Les mouvements | I- Interface menu du jeu   1. Traitement de l’image sur pygame 2. Chargement des fonctions et classes du jeu 3. Notion d’évènements   II- Le jeu   1. La notion d’Entity 2. Le fond et les plateformes 3. Le Joueur 4. Les mouvements 5. La collision 6. La gravité 7. La Caméra |

1. **Sources d’informations**

Pour nous aider à appréhender la bibliothèque Pygame (que nous n’avions jamais étudié), nous nous somme aidés du site Open Classroom, proposant un tutorial sur les bases de pygame (et python). De plus, nous avons lu différents projets similaires de pixel games fait par d’autres créateurs afin de mieux comprendre la logique et la forme requise pour un jeu vidéo. Il était aussi important à travers cette étape de se familiariser avec les fonctions importantes de pygame dans la création d’un jeu de plateforme (ou même un jeu tout court).

**Structure globale du projet :**



sur le personnage

(ensemble)

La gravité que subit le joueur

La collision entre le joueur et la plateforme

**II- Répartition des tâches et démarche collaborative :**

La répartition était en résumé ainsi :

|  |  |  |
| --- | --- | --- |
| **INES** | **SOLENA** | **DANA** |
| Interface du jeu (aka menu)  Appel d’un second code (Moteur\_SUPERDINO1) dans le code principal  Création de la plateforme dans la fonction | Aspect graphique (dessin du décor, de l’interface, de l’échantillon des sprites des personnages et de la plateforme)  Création de la plateforme dans le moteur  Travail de la collision  Travail de la gravité (dans le moteur) | Mouvement des personnages (dans le moteur de SUPER DINO 1 et SUPER DINO 2)  Travail de la caméra |

**Mode de collaboration :**

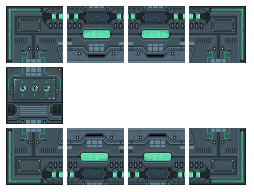
Afin de communiquer et modifier plus facilement notre programme codé en langage Python et avec l’aide de pygame, nous avons utilisé la plateforme Github qui permet d’interagir avec d’autres créateurs en cas de problèmes, et facilite l’ensemble de l’organisation du codage (car nous y avions tous accès simultanément).

Afin de finaliser notre projet, nous avons travaillé en dehors des heures de cours, fait de nombreuses recherches de notre côté afin d’avancer le projet. De plus, afin de mieux travailler, nous nous sommes vus le weekend pour mettre en commun nos travaux lors des moments où nous devions avancer et avons aussi utilisé Skype parfois.

**III- Réalisation personnelle**

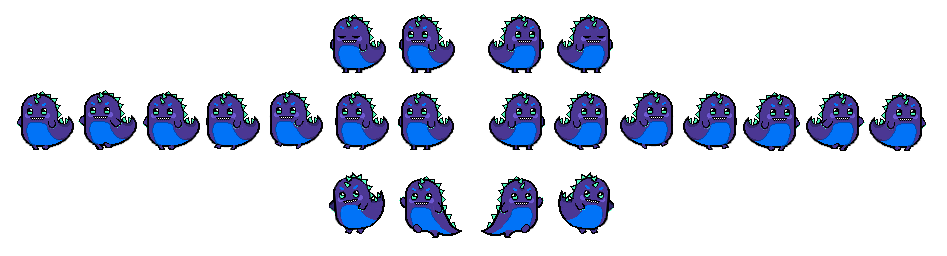
1. **L’aspect graphique**

Avant de présenter le code, je tenais à ne pas laisser de côté mon travail concernant les graphismes du jeu. En effet, je me suis chargée (en plus du codage) de la conception graphique de l’ensemble du jeu. Autrement dit, j’ai eu comme rôle de dessiner la totalité des éléments : fond du menu, du jeu principal, plateformes, échantillons des *« sprites »* (éléments graphiques pouvant se déplacer dans l’écran) du dinosaure. En voici des exemples :



**Le fond du jeu**

**Les plateformes**



J’avais dessiné un échantillon des différents mouvements du personnage car nous pensions au début donner l’illusion du mouvement selon la commande (haut, droite, etc…) du joueur. Nous voulions en d’autres termes le faire véritablement sauter, atterrir, marcher avec une animation. Cependant, par faute de temps, nous n’avons pas pu créer ces petites animations de mouvements.

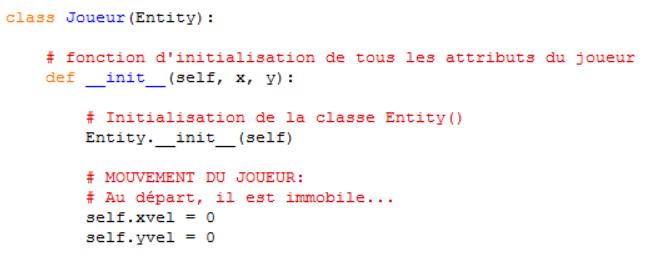
1. **Ma participation au code**

Bien qu’ayant participé à la création du premier jeu (Super Dino 1), je considère avoir pris une place plus importante dans la création du second jeu. Je vais donc expliquer ici les principes fondamentaux du code Moteur de SUPER DINO 2. Je laisse de côté l’explication du Code\_Principal\_SUPERDINO2 (Code du menu, faisant appel au Moteur pour faire marcher le jeu) en raison du fait qu’il soit pratiquement identique au Code\_Principal\_SUPERDINO1 que le dossier d’Inès présente avec détail.

**I- SUPER DINO 2 : Notions fondamentales : les classes, Entity() et l’instance « ensemble »**

1. **Notions de classes et fonctions**

Dans notre second jeu, nous avons fait appel à de nombreuses classes et fonctions. Il nous convient donc d’expliquer les bases de ces notions. La programmation orientée objet permet de créer des entités (objets) que l'on peut manipuler. Les objets peuvent interagir entre eux, cela facilite grandement la compréhension du code et sa maintenance.

Une **classe** regroupe des fonctions et des attributs qui permettent de définir un objet. Prenons comme exemple afin d’illustrer l’idée un extrait (simplifié et coupé) de la classe Joueur qui permet de créer notre joueur:

« **Méthode** », ou **fonction d’une classe**

Premier **paramètre** qui est associé à **l’objet/instance** en cours, pas encore déterminé





**Attributs** de la méthode \_init\_ de la classe Joueur

Autres **paramètres**, aussi appelés **arguments**

**Attributs** de l’objet

« class » représente ici la commande permettant de créer la classe, tandis que « def » permet de créer une fonction de cette classe. Dans notre cas, la classe Joueur est une sorte de modèle pour créer et gérer l’objet joueur.

1. **Fonctions ou méthodes**

Une **fonction** est une suite d'instructions que l'on peut appeler avec un nom. Elle peut s’apparenter à un dossier contenant un ensemble de commandes.

On appelle par ailleurs les fonctions d'une classe des **méthodes**. Ainsi, « mouvement » est une méthode de la classe Joueur.

Les **paramètres** (ou **arguments**) des fonctions et méthodes sont aussi importants. Ils se trouvent entre les parenthèses lors de la définition de la fonction. Dans notre cas, les paramètres de « mouvement » sont :

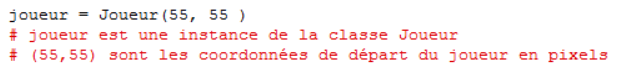
- self (l’instance/l’objet en cours, pas encore déterminé)

- haut, droite, ainsi que plateformes (un objet issu de la classe Plateforme, mais nous y reviendrons après).

CAS PARTICULIER : On remarque au sein des classes, que la fonction \_\_init\_\_ est spéciale. En effet, cette méthode est primordiale dans une classe (elle est toujours la première codée) et va être appelée lors de la création d'un objet (un joueur). Cette méthode est appelée un **constructeur.** Elle définit et initialise les **attributs** de la classe (éléments placés à droite de self.). Dans notre cas, il s’agit des principales *variables* du joueur, à savoir ici, xvel, sa vitesse sur l’axe des x et yvel, celle sur l’axe des y. Ici, on met par exemple la vitesse de base du joueur à 0 en x et y car il est immobile au départ.

1. **Objet ou instance**

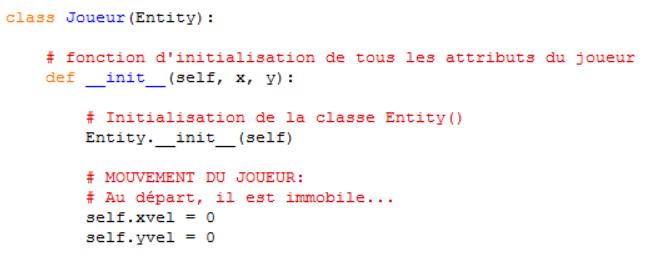
L’objet/instance d’une classe (qui va remplacer le paramètre « self » au sein de l’ensemble des fonctions de la classe) se crée de la façon suivante :



De ce fait, l’objet joueur de la classe « Joueur » est créé. Il est caractérisé par les attributs que lui confèrent la classe Joueur (en remplaçant « self ») et peut exécuter les fonctions/méthodes définies dans cette classe.

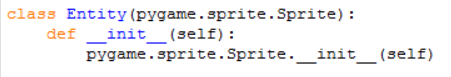
Qui plus est, la parenthèse (55, 55) à la suite de la classe Joueur signifie que les paramètres des fonctions/méthodes de la classe Joueurs (ici x et y) sont remplacés par 55 et 55, lorsque joueur est l’objet de la classe Joueur. Autrement dit, au sein de la classe Joueur, « joueur » remplace « self », x est remplacé par 55 et y est aussi remplacé par 55.

1. **Classe Entity()**

On remarque par ailleurs que, dans notre cas, la classe Joueur est définie avec l’argument Entity.

Cela signifie que la classe Joueur **hérite** de la classe Entity. Expliquons donc le principe de cette classe primordiale.

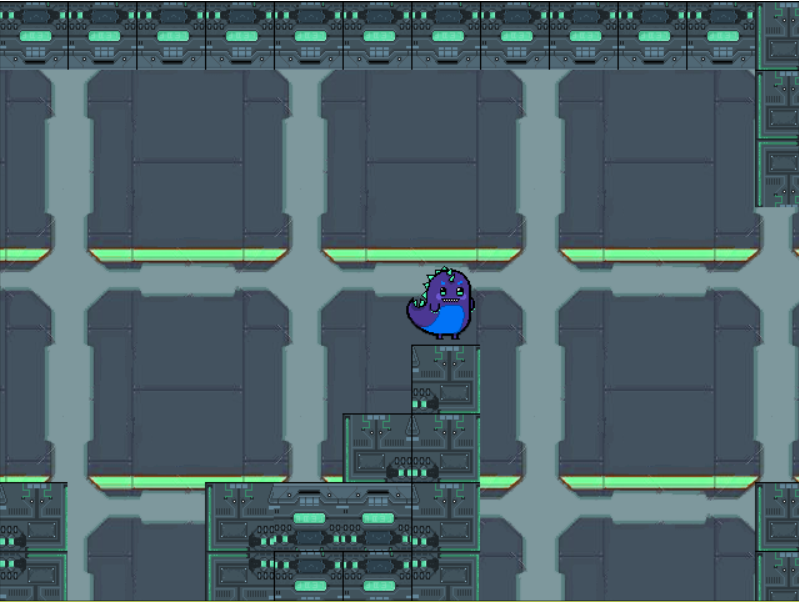
Conformément aux principes de la programmation orientée objet, Entity est une classe. Elle a la particularité d’hériter de **Pygame.sprite.Sprite**, une classe de base de Pygame utilisée pour « créer / caractériser » les objets visibles du jeu : les sprites. Ceux-ci représentent des rectangles texturés (avec une image).



De façon théorique, on peut concevoir notre interface et le niveau du jeu comme un ensemble de multiples images rectangulaires juxtaposées les unes à côté des autres. La classe Entity dans notre code permet de créer ces objets rectangulaires, ou sprites. Afin de simplifier la programmation, nous avons défini les objets visibles du jeu, dans presque tous les cas, comme des rectangles de 55 pixels\*55 pixels de côté. Le sprite est caractérisé par les coordonnées (x, y) du coin supérieur haut et gauche de son rectangle et par sa largeur et sa hauteur : (x, y, largeur, hauteur). Comme évoqué auparavant, dans presque tous les cas, la longueur et la hauteur de nos sprites font 55 pixels. Il est important de noter que les coordonnées de pygame suivent un repère inversé.

x

55 pixels



55

(x, y)

x

**largeur**



**hauteur**

y

**Coordonnées d’un**

**sprite(x, y, largeur, hauteur)**

y

**Découpage d’une partie du niveau**

La classe Entity est « **parent**» de toutes les classes servant à créer des objets visibles du jeu. Ainsi, dans notre code, les classes suivantes héritent d’Entity :

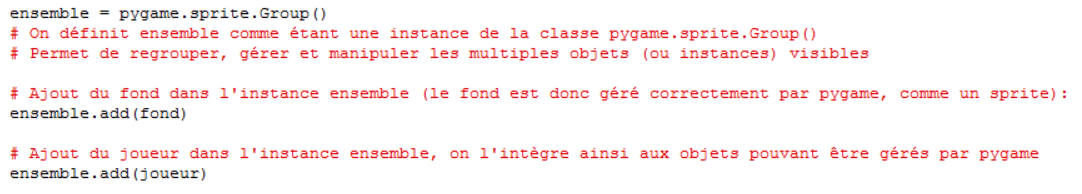
* **Joueur**  dont l’objet est le joueur (un sprite de 65\*65 pixels).

La classe Joueur(Entity) bénéficie donc de tous les attributs de Entity et en ajoute d’autres propres à elle-même.

* **Plateform** dont les objets sont les sprites représentant les blocs de structure du niveau, blocs entrant en collision avec le joueur. Cette classe charge l’image du bloc de structure qu’elle reçoit en argument dans son instance en cours (self), la redimensionne au format 55\*55 pixels et la caractérise comme un rectangle de coordonnées (x,y) de largeur 55 et hauteur 55 pixels
* **BlocVictoire** dont l’objet est le sprite du bloc d’arrivée que doit atteindre le joueur pour gagner. Cette classe charge l’image du sprite de sortie, un rectangle de couleur bleue et de surface 55\*55 pixels puis la caractérise comme un rectangle de coordonnées (x,y) de largeur 55 et hauteur 55 pixels
* **BlocMort** dont les objets sont les sprites représentant les zones entrainant la mort du joueur s’il les touche. Cette classe charge un rectangle de couleur rouge et de surface 55\*55 pixels puis la caractérise comme un rectangle de coordonnées (x,y) de largeur 55 et hauteur 55 pixels

1. **Instance/objet « ensemble »**

Afin de gérer efficacement tous ces objets visibles (Sprites), Pygame dispose d’une classe « container » appelée pygame.sprite.Group () qui permet de les regrouper dans un seul objet et de les manipuler. Dans notre code, l’objet issu de pygame.sprite.Group () est appelé « ensemble ».



Il est progressivement constitué de la totalité des objets visibles du jeu : le fond du niveau, les blocs de structure, les blocs de mort, le bloc d’arrivée et le joueur. Dans notre cas, le niveau finalisé s’étend sur 3740 pixel de largeur et 1100 pixels de hauteur.

Une fois finalisé, l’objet « ensemble » n’évolue que très peu au cours du jeu. Seule, la position du bloc « joueur » (au sein de « ensemble ») change en fonction du mouvement du joueur et de ses collisions avec les blocs de structure.

De plus, un des intérêts de l’objet « ensemble » est lié au fait qu’il permette à l’ensemble des sprites du jeu d’être affiché avec une seule boucle :



En effet, ici, pour tout élément (sprite) de l’objet « ensemble » (regroupant donc le fond, les plateformes, le joueur…), Pygame vient coller l’image de chaque sprite à l’écran (on dit que l’image est « blitté » à la fenêtre de l’écran) à travers la commande **.blit(),** et vient afficher à l’écran la zone du niveau autour du joueur sur une surface de destination définie par l’exécution de la méthode « appliquer » de la classe Camera (cette notion sera d’avantage développée par Dana).

NOTE : « fenetre » représenté l’écran d’affichage crée au tout début du code à partir d’une autre commande Pygame : pygame.display.set\_mode (taille de la fenêtre d’affichage).

**II- SUPER DINO 2 : Déroulement du code principal - la fonction main()**

1. **Son utilité**

Un autre élément très important pour comprendre l’ensemble du code est la fonction main(). Celle-ci représente la fonction primordiale du jeu, celle contenant l’ensemble des actions nécessaires au lancement de la partie. On retrouve donc au sein de cette fonction :

- La création de l’ensemble des objets (fond, joueur, camera…) issus des différentes classes du jeu

- La création de l’objet « ensemble » regroupant l’ensemble des objets du jeu, afin de les blitter (coller, afficher) à l’écran en fonction de la caméra à la fin de cette fonction main().

- La création entière du niveau et des plateformes (le concept sera expliqué ultérieurement dans le dossier) à l’aide de la classe Plateforme, BlocVictoire et BlocMort.

- La boucle principale du jeu, permettant au jeu de tourner indéfiniment (grâce au « while 1 : ») sauf si l’on clique sur la croix de la fenêtre, ou que l’on appuie sur ECHAP. La boucle principale vient aussi à chaque cycle:

* Limiter le nombre de cycle par seconde avec la méthode tick (nb de cycles par seconde) de pygame
* Attribuer les évènements des touches aux variables de mouvement (haut, gauche, droit) en fonction de la touche appuyée par l’utilisateur, les actions de l’utilisateur étant contrôlées par la fonction pygame.event.get() de Pygame
* Appliquer les mouvements du joueur en fonction des variables de mouvement et des blocs de structure:
* Actualiser la caméra en fonction du déplacement du joueur de sorte qu’elle « suive » le joueur par application de la méthode « mouvements » de la classe Camera
* Afficher chaque sprite de l’objet « ensemble » grâce à la boucle for vu auparavant.

La fonction main() correspondant à l’ensemble du jeu est donc au final utilisé dans Code\_Principal\_SUPERDINO2 (le menu du jeu) afin de lancer la partie.

Ici, je n’expliquerai pas tous les éléments de la fonction main() en détail, mais plus particulièrement ceux auxquels je me suis le plus occupée : Le fond et le niveau.

1. **Le fond et le niveau**
2. **Le fond**

Le fond se crée dans la fonction principale du jeu : la fonction main().

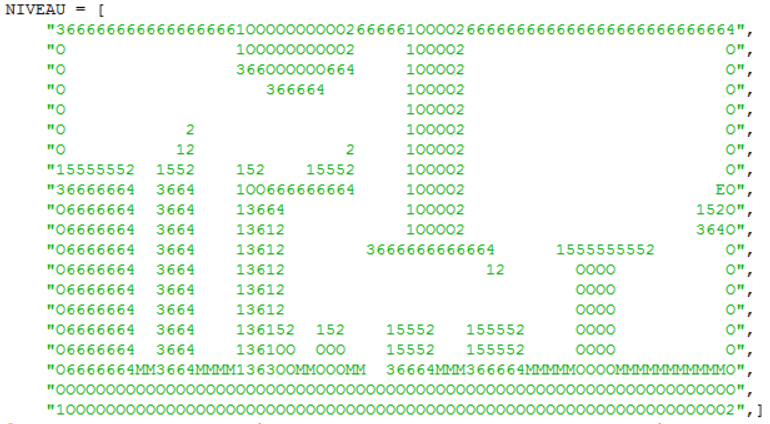
- Après avoir créé le fond en tant qu’objet de Entity(), nous le chargeons à l’aide de la commande de Pygame « pygame.image.load(-emplacement et nom de l’image-).convert() ».

- Afin de pouvoir utiliser le fond dans pygame, nous le définissons comme un rectangle dont son coin gauche est de 0 et qui s’étend jusqu’à la largeur de l’image et la hauteur de l’image.

- Comme nous l’avons vu un peu avant, le fond est ensuite intégré à l’objet « ensemble », puis « blitté » (collé à l’écran) à la fin de la fonction main, comme le reste des éléments de l’instance « ensemble ».

1. **La création du niveau et des plateformes**

Ensuite, on crée le « NIVEAU », une liste de 20 chaines de 68 caractères chacunes. Une liste représente une variable au sein de laquelle on peut mettre plusieurs variables. Ces 68 caractères correspondent à un bloc de structure de 55\*55 pixels dont l’image (le type) sera définie en fonction des caractères 0, 1, 2, 3, 4, 5, 6, M ou E contenus dans la liste NIVEAU. La création d’une telle liste nous permet d’avoir un aperçu très « graphique » de notre niveau.



1er chaine/item de la liste :

NIVEAU[0]

NIVEAU[1]

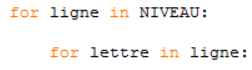
Etc….

On crée ensuite une liste vide « plateformes » qui sera progressivement remplie par les objets (sprites) résultant de la « traduction en sprites» des caractères contenus dans NIVEAU.

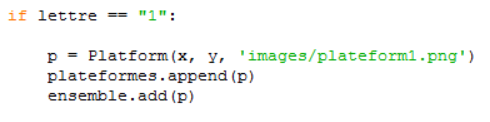
En effet, la création du niveau consiste à tout d’abord assimiler chaque caractère de chaque item de la liste « NIVEAU » à un bloc imagé (égal à une instance plateforme p), afin de les ranger un par un dans la liste plateformes (utilisé ultérieurement pour la collision) ainsi que dans l’objet « ensemble ». Pour cela :

On initialise les coordonnées x et y à 0, ces coordonnées seront celles du premier bloc de structure du niveau.

On lance ensuite deux boucles « for » qui permettent d’associer en premier lieu chaque caractère contenu dans un item (soit ici une ligne horizontale) du NIVEAU à un sprite de 55\*55 pixels, puis de répéter l’action sur une ligne plus basse (donc au second item, et ainsi de suite).



En quelque sorte, on traduit chaque caractère en un sprite et ce, pour tous les caractères. Ainsi, pour chaque caractère de NIVEAU différent de « M » ou « E » :



* On crée une instance de la classe Platform, un sprite (nommé p), ayant comme arguments les coordonnées x et y, ainsi que l'image du bloc correspondant au caractère en cours de traduction.

La classe Plateform sert de son coté à charger l’image du bloc et caractériser l’objet comme un bloc rectangulaire texturé de coordonnées précises x et y : un sprite.

* On ajoute ce sprite dans la liste "plateformes" selon la méthode List append de Python. De la sorte, les blocs de structure s'ajoutent progressivement dans « plateformes » (la liste des objets qui entrent en collision avec le joueur)
* On ajoute ce sprite à l’objet « ensemble » selon la méthode add de pygame.sprite.Group(). De la sorte, les blocs de structure s'ajoutent progressivement aussi dans « ensemble » (les objets visibles du jeu)

Ensuite, par exactement le même principe, si le caractère est respectivement M ou E, on crée une instance de la Classe respectivement BlocMort ou BlocVictoire. Cette instance est aussi ajoutée à la liste « plateformes » et à l’objet « ensemble ».

Après chaque assimilation, au sein de la boucle, la valeur x augmente de 55 (largeur d’un sprite) pour modifier les coordonnées de la plateforme crée, de façon à ce que la nouvelle soit à droite de l’ancienne. Cette action est répétée selon le nombre de caractères dans une chaine. Puis, lorsque le dernier caractère de la chaine est traduit en sprite, la seconde boucle est finie, et la première recommence avec cette fois-ci y = y + 55, signifiant que l’on descend d’une ligne. De plus, on remet x = 0, ce qui permet de recommencer la même opération, mais seulement une ligne plus bas en partant de la gauche.

On peut schématiser l’opération par l’image (simplifiée) suivante :

NIVEAU : [

Etc…

X += 55

X += 55



Y +=55

X = 0

Après la plateforme et le niveau, je me suis occupée de résoudre le problème concernant le joueur (face au fait qu’il nous manquait dans notre premier code la gravité et le système de collision).

**III- Le joueur : la classe Joueur()**

La Classe Joueur permet de créer et gérer le joueur, ses mouvements et ses collisions avec les blocs de structure (listés dans « plateformes »). Elle hérite de la classe Entity, ce qui permet à « joueur », l’objet qui en est issu, d’être un sprite (un objet visible). Elle nécessite comme arguments les coordonnées x, y du joueur (l’objet qu’elle instancie).

Ses attributs sont :

* xvel, sa vitesse sur l’axe des x
* yvel, sa vitesse sur l’axe des y
* onGround, une variable logique caractérisant le fait que le joueur est sur le sol ou non
* image, son image
* rect, son rectangle de coordonnées (x, y) et de largeur la taille du personnage et de hauteur la taille du personnage, soit 65\*65 pixels

Cette classe possède deux méthodes :

1. **Mouvements (self, haut, gauche, droite, plateformes) : la gravité**

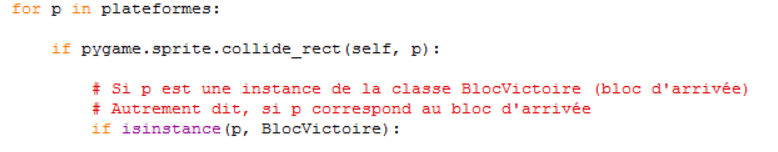
Cette méthode gère les mouvements du joueur. Elle exécute le déplacement du joueur en fonction notamment des variables logiques haut, gauche, droite, lesquelles dépendent des touches saisies au clavier. Par exemple, si la variable logique haut est vraie, la vitesse du joueur sur l’axe des x, xvel, sera fixée à 10 pixels par cycle. (L’idée de mouvement du joueur sera plus approfondie par Dana).

De plus elle gère la **gravité** : si le joueur est dans les airs (onGround est faux, le joueur n’est pas sur le sol), sa vitesse sur l’axe des y, yvel, sera augmentée de 0,3 pixels par cycle (accélération). Ainsi, lorsque le joueur n’est plus sur le sol et qu’il saute, il est soumis à la fois à une montée de 10 pixels par 10 pixels (sur l‘axe des y) et une force qui le fait descendre de 0,3 pixels par 0,3 pixels. Ainsi, plus le joueur reste dans les airs, plus la valeur de y se rapproche de 0, jusqu’à s’annuler totalement (au moment où le personnage reste dans les airs, immobiles, un court instant), pour enfin devenir positif et amener donc à la chute du personnage. De cette façon, on est parvenus à simuler une force de gravité sur le joueur.

Ensuite, la fonction n’autorise le saut du personnage que s’il est sur le sol (onGround vrai). Enfin, elle gère les collisions du joueur avec les plateformes. Elle exécute pour cela la méthode collision présentée ici :

1. **Collision (self, xvel, yvel, plateformes)**

Cette méthode détecte les collisions du joueur avec les blocs de structure (listés dans « plateformes ») et agit en conséquence (actualise les variables ou lance l’écran de la victoire ou de la mort). Pour détecter la collision, elle fait appel à une fonction de pygame, **pygame.sprite.collide\_rect(self, p),** qui détecte les collisions de sprites rectangles.



Ainsi, pour chaque sprite « p » provenant de « plateformes », Pygame détecte (à l’aide de la commande ci-dessus) la collision entre « self » (qui deviendra l’objet joueur) et la plateforme p. L’action faisant arrêter le personnage contre le bloc lors d’une collision se fait de la façon suivante :



Ici, dans le cas où la variation de position du personnage xvel est positive (donc lorsque le joueur va vers la droite), et que l’on a collision, alors le côté droit du rectangle du joueur « self.rect.right » revient au côté gauche du rectangle de la plateforme « p.rect.left ».



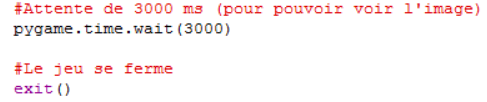
=

Le personnage ne peut alors plus bouger à droite. On répète le principe pour chaque côté du rectangle du personnage avec le coté de plateforme correspondant, et le tour est joué !

Ensuite, afin de détecter une collision entre le joueur et un bloc issu de la classe BlocVictoire, nous avons utilisé la commande pygame suivante :



Cette commande « isinstance » signifie que si le bloc de plateforme p avec lequel le joueur est en collision est une instance de la classe BlocVictoire, alors (suite des évènements) un écran de victoire apparait, puis la fenêtre se ferme après 3 seconde d’attente. Pour ces actions, des commandes pygame sont de nouveaux utilisées :



Selon le même principe, en cas de collision avec un bloc de mort (plateforme devenue une instance de la classe BlocMort), la méthode affiche l’écran « game over » et relance le jeu après 3 secondes.

**IV – Problèmes, erreurs et résolutions**

Au cours de notre code, au vue de la complexité de notre projet, nous nous sommes confrontées à de nombreuses impasses. En effet, l’existence de ce second code et jeu est en soi d’ors et déjà une réponse à nos impasses. En effet, dans notre première version du jeu, nous avions de nombreux problèmes :

Après avoir réussi à blitter notre fond et nos plateformes, nous avions des problèmes quant à l’apparition du personnage et sa mise en mouvement (en effet, on demandait à Pygame de charger l’image alors qu’aucun personnage ne s’affichait à l’écran). Après avoir eu recours aux classes afin d’ordonner les mouvements du personnage, le résultat était devenu pire. A partir de ce moment-là, nous avons conclu que nous n’étions pas assez familières avec le vocabulaire des classes, ainsi que les commandes de pygame. De ce fait, nous avons pris la décision de nous diviser momentanément afin de trouver des solutions au problème : D’une part, Dana et moi sommes allées trouver des réponses sur internet, en recherchant sur différents forums des personnes qui recherchaient les mêmes éléments que nous (problème des mouvements, et caméra qui suit le personnage par exemple), tandis qu’Inès (et Dana pour les mouvements) essayaient de trouver une solution face à notre premier code. Après avoir résolu le problème du premier code (résolution qui sera expliquée dans le dossier d’Inès ou de Dana), nous avons obtenu un code qui marchait. Cependant, nous avons au final obtenu un personnage qui se déplace devant un fond, sans être soumis à la gravité, qui n’interagit entièrement pas avec son environnement (les plateformes) et dont la caméra ne le suit pas. Ce résultat ne représentant pas un jeu vidéo complet, nous avons ensuite voulu repartir de 0 et chercher des réponses dans des forums. C’est de cette façon que nous avons trouvé un code moteur (une sorte de squelette) comprenant l’ensemble des éléments qui nous manquaient (caméra, la collision et la gravité). En suivant les principes de ce code, nous avons réussi (après des nombreuses heures et journées de travaux en équipe) à comprendre la logique (peu évidente lorsque l’on n’est pas familier avec l’ensemble des commandes Pygame) derrière la création d’un code orienté entièrement Objet et sommes parvenues à suivre les mêmes procédés pour l’usage de notre code.

Les éléments qui manquaient dans notre premier code étaient premièrement reliés à notre manque de connaissances sur les classes, ce qui rendait nos manœuvres très réduites car lors d’un problème, nous avions des grandes difficultés à comprendre la source du problème. Puisque nous ne connaissions pas bien les classes, nous ne connaissions pas le principe de la classe Entity() (permettant de créer des sprites). Sans cette classe, les objets ne pouvaient pas être créés convenablement. De même pour l’instance « ensemble » permettant de gérer et regrouper les sprites du niveau. Maîtriser la notion de classes était primordial dans ce genre de code ou un nombre important d’éléments (objets et classes) sont en jeux et dépendent les uns des autres. Heureusement, grâce à nos recherches sur le sujet, étudier avec persévérance le code « squelette » nous a permis d’atteindre une grande partie de notre objectif.

Un regret restant serait de ne pas avoir eu le temps de pouvoir faire des petites animations quand le personnage bouge. Mais l’étude de ce code m’a été dans tous les cas très bénéfique, car l’étudier intensément m’a permis sous un temps record de comprendre tout ce que je n’avais pas compris. Grâce à l’étude de ce code, je me sens désormais capable de créer un jeu vidéo de plateforme en utilisant python orienté Objet.

**V- Bilan et perspectives**

Si nous avions eu plus de temps, nous aurions pu développer plus le graphisme et ajouter des effets sonores afin de renforcer l’expérience pour le joueur et parfaire le jeu. Le travail en équipe a été très enrichissant car nous avons pu confronter nos différentes idées, rencontrer des obstacles et faire évoluer nos méthodes de travail : se mettre d’accord sur une grande ligne, se répartir certaines tâches, les mettre en commun et s’aider mutuellement quand cela a été nécessaire. De plus, notre équipe était assez complémentaire et j’ai beaucoup apprécié de travailler avec mon groupe sur un projet aussi créatif. Le travail en mode projet est aussi intéressant bien que source de stress car nous avons choisi un sujet pour lequel nous avions un peu sous-estimé les difficultés et la charge de travail. Ainsi, nous avons dû le modifier afin de respecter les délais, ce qui n’empêchera pas plus tard de continuer le jeu. Cette expérience m’a appris l’importance de faire un rétro-planning afin de prévoir les différentes étapes. Au final, ce codage de jeu vidéo a été très intéressant et bénéfique. Ce qui m’a plus dans ce projet est de mêler l’aspect de programmation à celui du graphisme et de la créativité. Ainsi, travailler sur un projet créatif que nous avons tous choisi de créer ensemble m’a beaucoup motivée et m’a amené à me dépasser. De plus, cela m’a conforté dans l’idée de travailler plus tard dans un domaine où l’animation, la création et le jeu vidéo seront importants.

**VI – Annexe**

Lien Github :

<https://github.com/ecolelasource92/Super-Dino/tree/master/Code%20global/Super-Dino>

Sources d’informations utilisées :

<https://jeux.developpez.com/tutoriels/python/sdl/Pygame/introduction-a-pygame/>

Pygame documentation :

<http://www.pygame.org/docs/>

Apprendre à programmer avec Python :

<https://python.developpez.com/cours/TutoSwinnen/>

Apprenez à programmer en Python :

<https://openclassrooms.com/courses/apprenez-a-programmer-en-python>

Vidéos utilisées :

<https://www.youtube.com/watch?v=bxibfL4TjWo&t=6s>

<https://www.youtube.com/watch?v=mTmJfWdZzbo>