CY-TECH PYTHON GAMING
Développement de jeu d'arcade en langage Python

Romuald GRIGNON

CY-Tech
Site de Cergy-Pontoise
CY Cergy-Paris Université

Année universitaire 2021-2022

PRESENTATION

Contexte

- Dans cet atelier, nous allons voir comment utiliser un moteur de jeu 2D pour créer un petit jeu d'arcade en utilisant les périphériques de l'ordinateur (clavier, contrôleur), en affichant des éléments graphiques animés et en jouant des sons associés aux actions du joueur
- Le langage utilisé sera le langage Python 3
- La bibliothèque de jeu 2D utilisée est : « **arcade** » (**https://arcade.academy**), développée par Paul V. CRAVEN (Professor, Simpson College, Indianola, Iowa, USA)
- Cette bibliothèque utilise la programmation dite « orientée objet ». Si vous n'êtes pas familier avec cette manière de programmer, ce n'est pas un problème : tout votre jeu pourra être codé de façon procédurale. Seule la syntaxe pour l'utilisation de vos variables et vos fonctions pourra différer légèrement par rapport à ce que vous avez vu précédemment
- Pour les plus chevronnés d'entre vous, il pourra être intéressant de coder votre jeu avec une orientation objet malgré tout, mais le but premier est d'utiliser les connaissances en algorithmique procédurale vues précédemment pour créer un jeu vidéo minimaliste mais fonctionnel

Méthode de développement

- Cet atelier sera totalement libre et vous n’avez donc aucune contrainte de temps ou d’objectif. Le résultat attendu est celui que vous fournirez tout simplement
- Néanmoins, pour s’assurer que des participants isolés n’abandonnent pas face aux difficultés, il est prévu des rendez-vous réguliers pour faire le point sur les avancements de chaque projet et éventuellement travailler ensemble sur la résolution d’une problématique technique ou trouver de nouvelles idées de gameplay ou de nouvelles manières plus astucieuses, plus efficaces, de coder des fonctionnalités
- Le but est de garantir à chacun, à son niveau, la possibilité de réaliser un jeu complet, même minimaliste
- L’etat d’esprit de cet événement est donc basé sur le partage de connaissances, sur l’application de ce qui a été vu dans les différents cours, dans le but de réussir à coder un jeu fonctionnel : il n’est donc pas question de faire un concours ici

Type de jeu à créer

- Vous pouvez créer n’importe quel type de jeu mais il est conseillé de partir sur un jeu avec un minimum de fonctionnalités
- Essayez de trouver des idées de jeux d’arcade existants, sur un seul écran fixe afin de limiter les problématiques, surtout si vous débutez. La gestion de caméra reste quelque chose de possible mais toute fonctionnalité ajoutée à votre jeu entraîne une gestion technique supplémentaire
- Essayez de limiter les fonctionnalités de manière à terminer votre jeu, et il sera toujours possible d’ajouter des évolutions si il vous reste du temps
- Les jeux multi-joueurs compétitifs ou coopératifs sont toujours des solutions de choix car elles ajoutent une dimension conviviale
- Il n’y a pas de module « réseau » fourni dans le code du projet, ce qui signifie qu’il faudra privilégier un jeu local.
- Dans le cas d’un jeu multi-joueurs, il est donc conseillé de partir sur un jeu dans lequel tous les joueurs peuvent avoir la même vue

Type de jeu à créer

- Quelque soit le type de jeu et la quantité de fonctionnalités implémentées, certaines étapes sont quasi-incontournables :
  - Plusieurs scènes qui se succèdent (écran d'accueil, sélection du joueur, phase de jeu, page de pause, écran de fin de jeu) ou au minimum une gestion de l'état du jeu pour savoir quoi afficher
  - Un personnage/véhicule (animé ou non) qui peut être déplacé par le clavier, la souris, un gamepad
  - Un décor de fond statique, ou un fond animé (vous pourrez coder un fond de type « parallax ») avec éventuellement des éléments animés
  - Une gestion de collisions entre les différents éléments
  - Une gestion et un affichage de variables diverses telles que les points de vie, le score, les munitions, le temps, ..., variables qui peuvent être utilisées pour définir les critères de fin de jeu
  - Une gestion des sons pour ajouter une dimension sonore à votre jeu

Type de jeu à créer

- Il existe également une dimension qui semble implicite mais qui nécessite de la charge de travail : les éléments graphiques
- Il existe plusieurs sites spécialisés dans la fourniture de ressources graphiques gratuites à l'utilisation : n'hésitez pas à les utiliser
- Même en récupérant des images sur Internet il sera souvent nécessaire de les détourer/modifier pour qu'elles s'adaptent à votre projet
- Si vous avez des camarades qui ont un certain talent dans l'infographie, il serait intéressant d'essayer de les motiver à vous rejoindre de manière à collaborer
- Si vraiment vous n'avez aucun talent graphique, aucune personne de votre entourage pour vous rejoindre dans ce projet, et que vous ne vous sentez pas prêts à manipuler des ressources recupérées sur le net, essayez de concevoir votre jeu autour de formes géométriques (votre personnage est un carré, les ennemis des cercles, …) : tous les jeux de rythme ne nécessitent pas forcément de sprites évolués : mettez l'accent sur le gameplay !

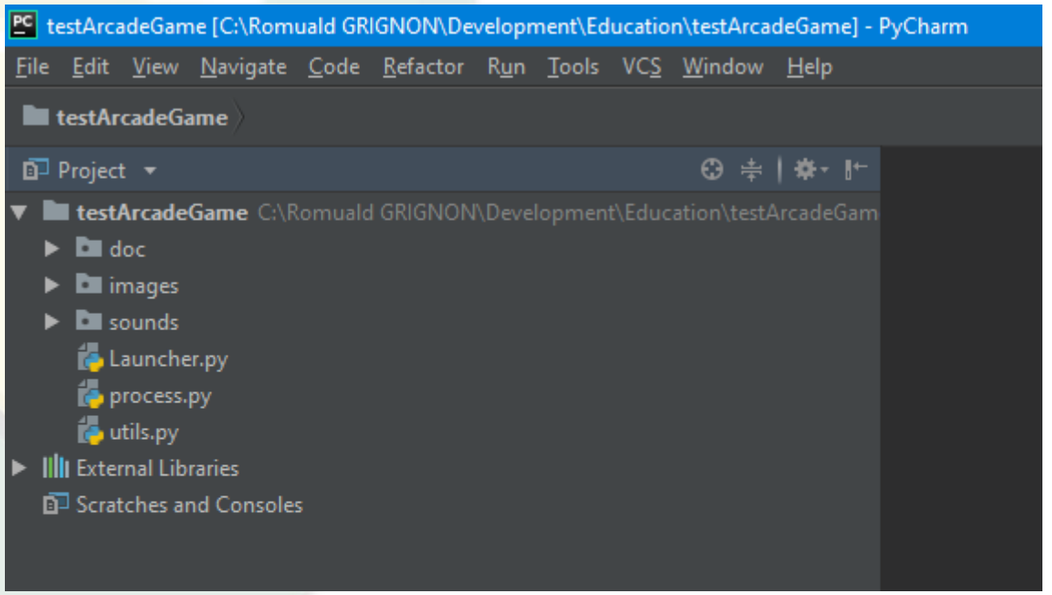
ENVIRONNEMENT DE TRAVAIL

Environnement de travail

- Nous allons utiliser le logiciel PyCharm de la société JetBrains
- Cet outil est un environnement de développement qui permet d'avoir sur le même écran, la liste des fichiers du projet sur le disque dur, la fenêtre d'édition de nos fichiers de code, la sortie du terminal, et des boutons permettant d'automatiser des tâches, comme par exemple lancer le jeu (il tourne sous Windows, Linux et MacOs)
- Nous allons également vous fournir des fichiers pour démarrer cet atelier. Ces fichiers constituent le squelette de votre programme, les différents endroits du code dans lesquels vous pourrez agir seront listés dans les diapositives suivantes
- Dans les fichiers fournis, se trouvent également des fonctions permettant de vous simplifier le travail pour gérer tous les aspects graphiques et sonores du jeu, mais aussi pour gérer tous les périphériques tels que des claviers, souris ou contrôleurs de jeu USB, ou gérer des collisions
- Avant de commencer la présentation du moteur de jeu et du code fourni, il va falloir installer et configurer les outils et vérifier que tout cela tourne sur votre machine

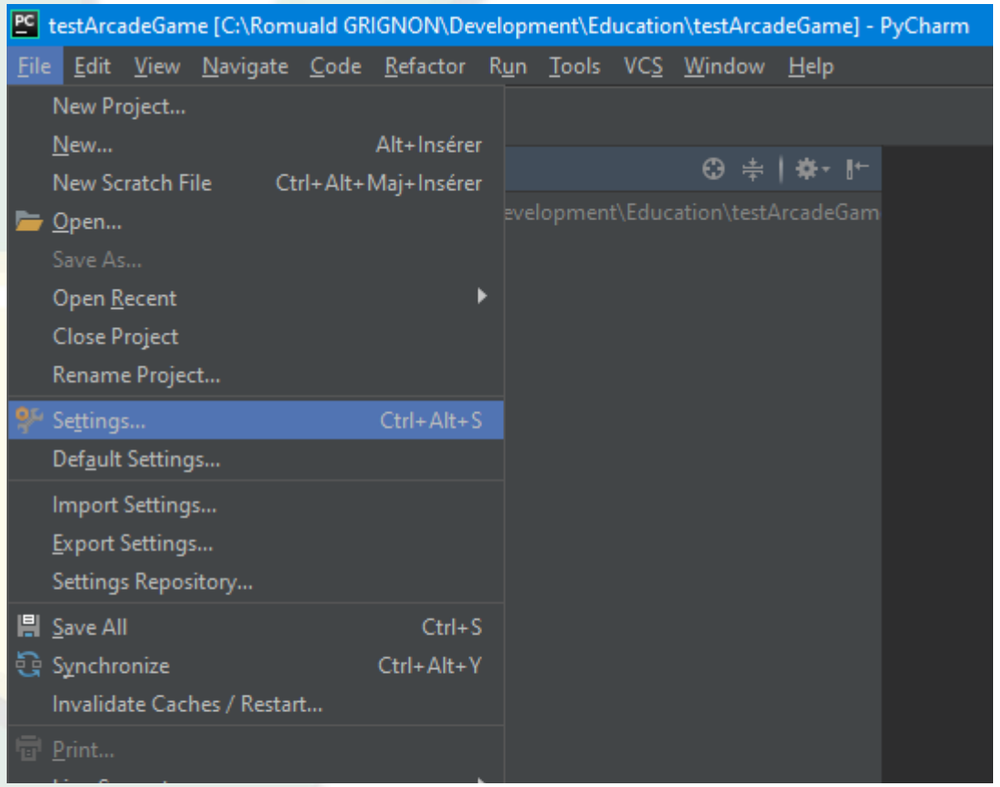
Environnement de travail

- Après voir installé et lancé le logiciel PyCharm (la version gratuite « Community » suffit), il vous faudra décompresser les fichiers fournis dans un répertoire de travail que vous aurez créé pour cet atelier
- Depuis le logiciel PyCharm, sélectionner l'option « open », puis choisissez le dossier dans lequel vous avez décompressé vos fichiers sources.
- Vous verrez alors apparaître la fenetre de votre projet dans le bandeau vertical de gauche. C'est à cet endroit que vous pouvez voir et accéder à tous vos fichiers



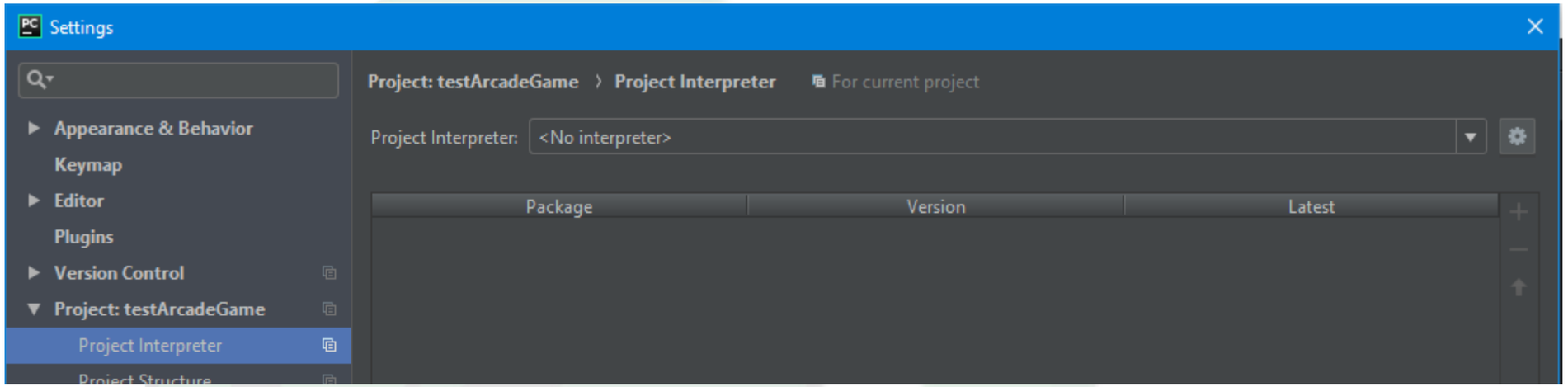
Environnement de travail

- Le code qui vous a été fourni utilise le code de la bibliothèque « arcade ». Il faut donc faire en sorte de récupérer cette bibliothèque sur votre machine : pour cela nous allons créer ce que python appelle un « environnement virtuel » (venv) sur votre machine, qui est en fait un dossier contenant l'ensemble du noyau Python et des bibliothèques à utiliser
- Allez dans le menu File → Settings

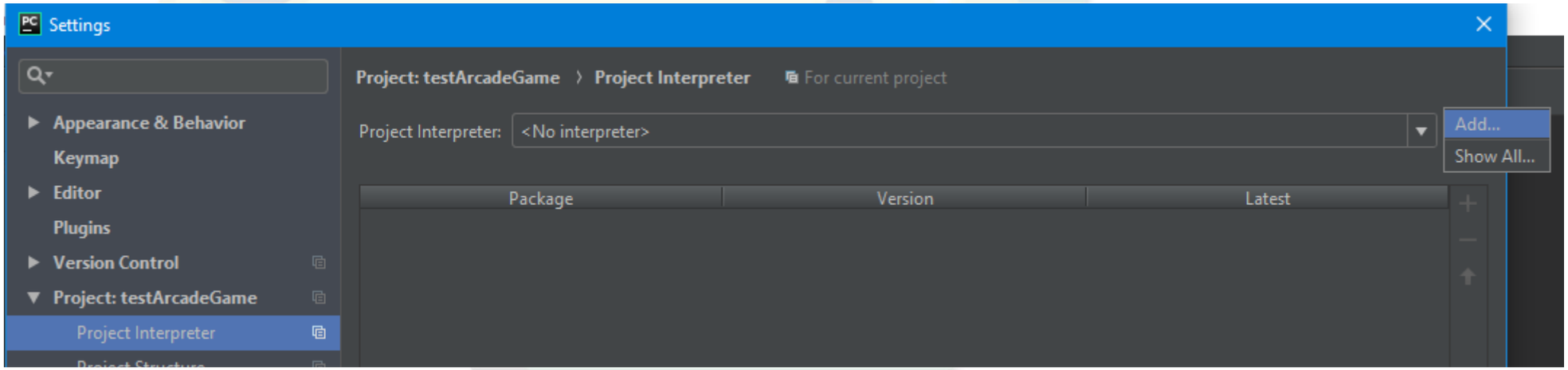


Environnement de travail

- Sélectionnez l'option « Project Interpreter »

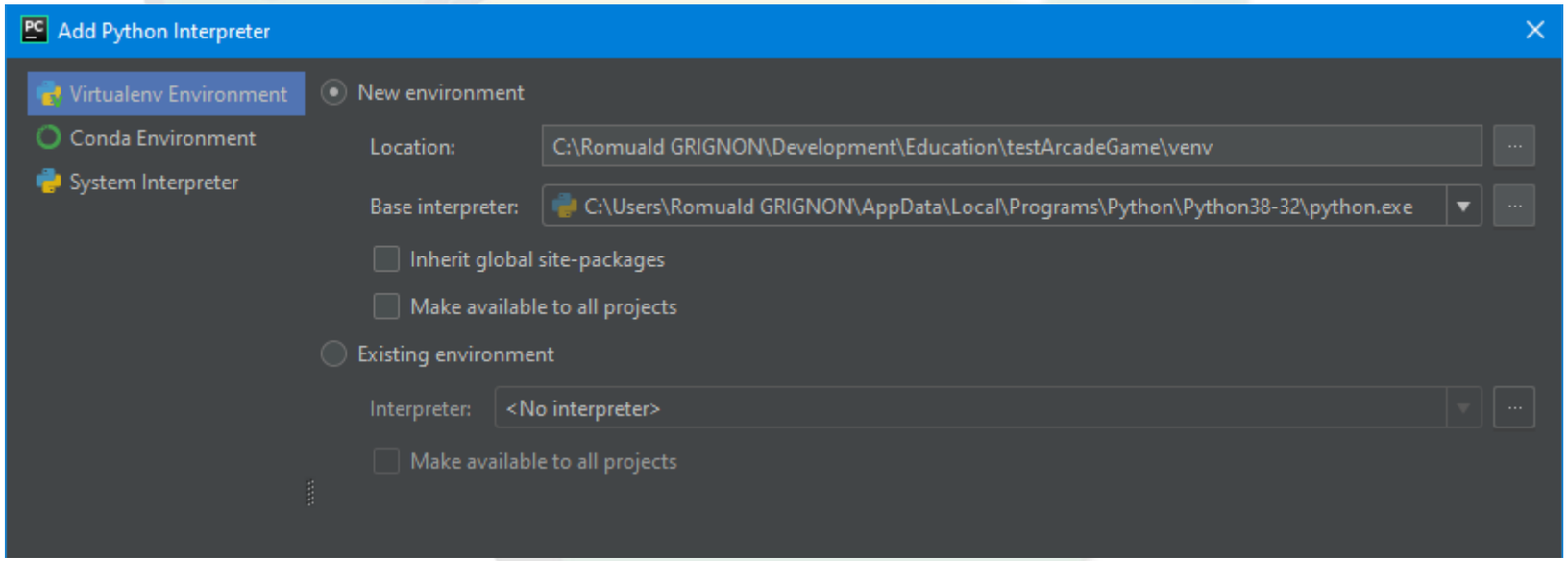


- Puis cliquez sur l'engrenage en haut à droite, puis sur « Add », pour créer un nouvel espace pour stocker l'interpréteur Python et les bibliothèques à utiliser



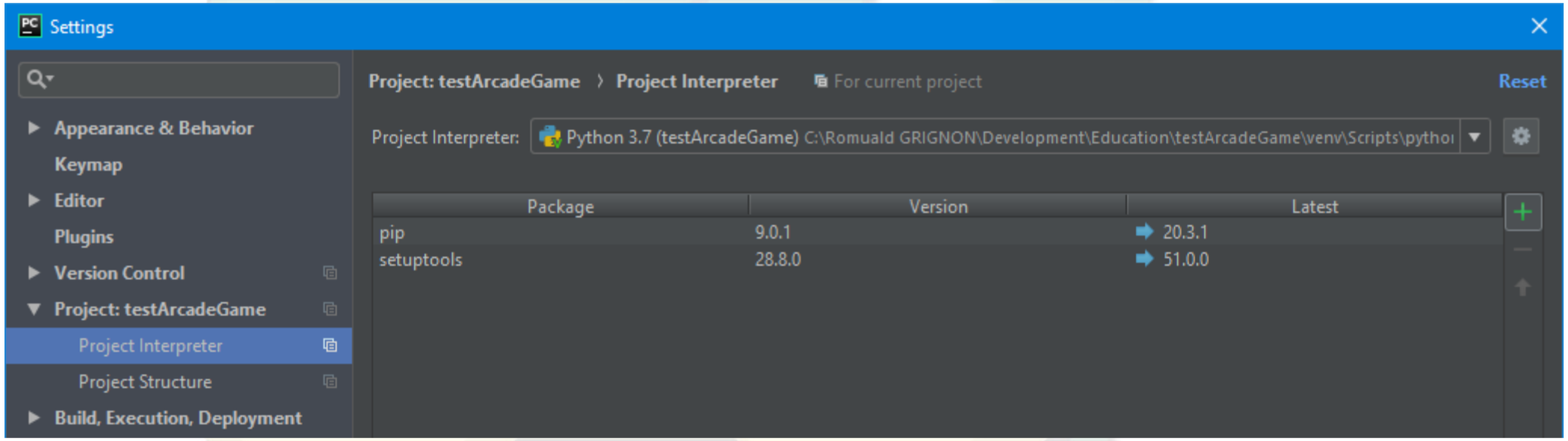
Environnement de travail

- Dans la fenêtre qui s'ouvre, choisissez « Virtual Environment », puis « New Environment »
- Dans l'option « Location », indiquez un sous dossier de votre dossier de travail, et appelez-le « venv »
- Dans l'option « Base Interpreter », entrez le chemin où se trouve l'exécutable Python sur votre machine (ATTENTION, à la date de rédaction de ce document, il semble que Python3.10 ne fonctionne pas avec le code qui est fourni, il est conseillé de prendre Python3.8 ou moins)
- Ensuite vous pouvez valider



Environnement de travail

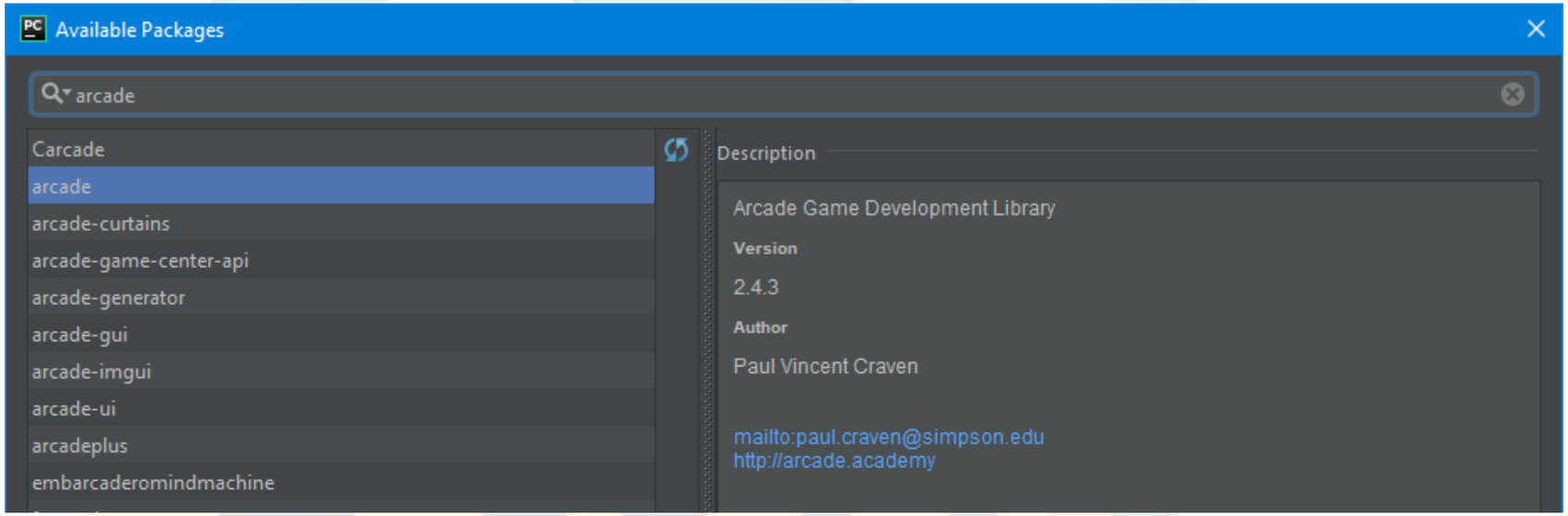
- Maintenant que le dossier d'environnement virtuel est créé, il va falloir télécharger la bibliothèque arcade
- Avant de procéder à cette étape, assurez-vous d'avoir la dernière version de la bibliothèque « pip » affichée dans votre venv.



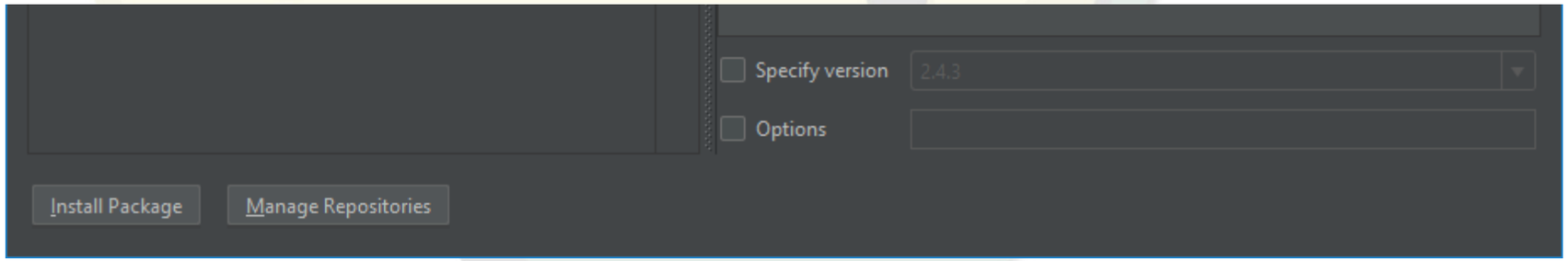
- Ensuite cliquez sur la croix verte (signe '+') en haut à droite de la fenêtre pour ajouter une nouvelle bibliothèque à notre environnement

Environnement de travail

- La fenêtre de recherche apparaît : entrez le nom « arcade » dans la barre de recherche tout en haut, puis sélectionnez la ligne où le nom de « Paul Vincent Craven » apparaît comme auteur. Specifiez la version de « **arcade** » comme étant la **2.4.3**

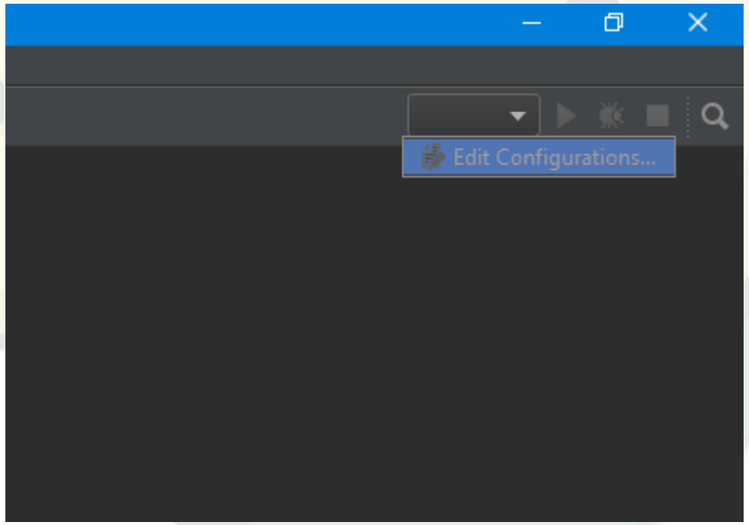


- Cliquez ensuite sur le bouton « Install Package » en bas de la fenêtre pour lancer l'installation



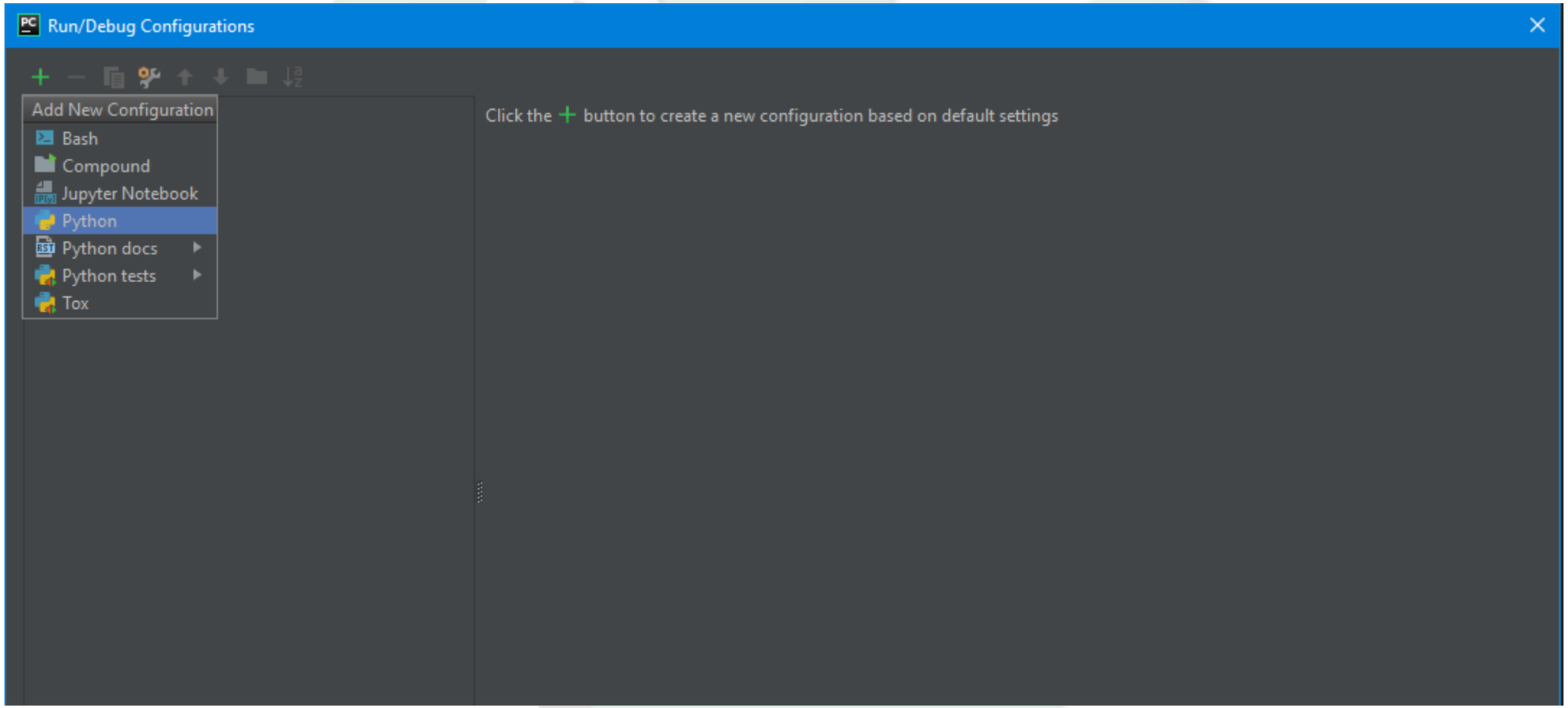
Environnement de travail

- Une fois l’installation de arcade terminée, cliquez à nouveau sur le ‘+’ pour rechercher une bibliothèque : recherchez **PyMunk** et installez la version specifique **5.7.0**
- Une fois l'installation terminée avec succès, il nous reste maintenant à créer ce que l'on appelle une « configuration » dans PyCharm
- Les configurations permettent de préparer le lancement d'un programme Python avec des paramètres spécifiques. Cela permet d'avoir plusieurs modes d'execution directement accessible en 1 clic.
- Cliquez sur la liste déroulante en haut à droite de la fenêtre de PyCharm et choisissez le seul choix accessible pour le moment : « Edit Configurations »



Environnement de travail

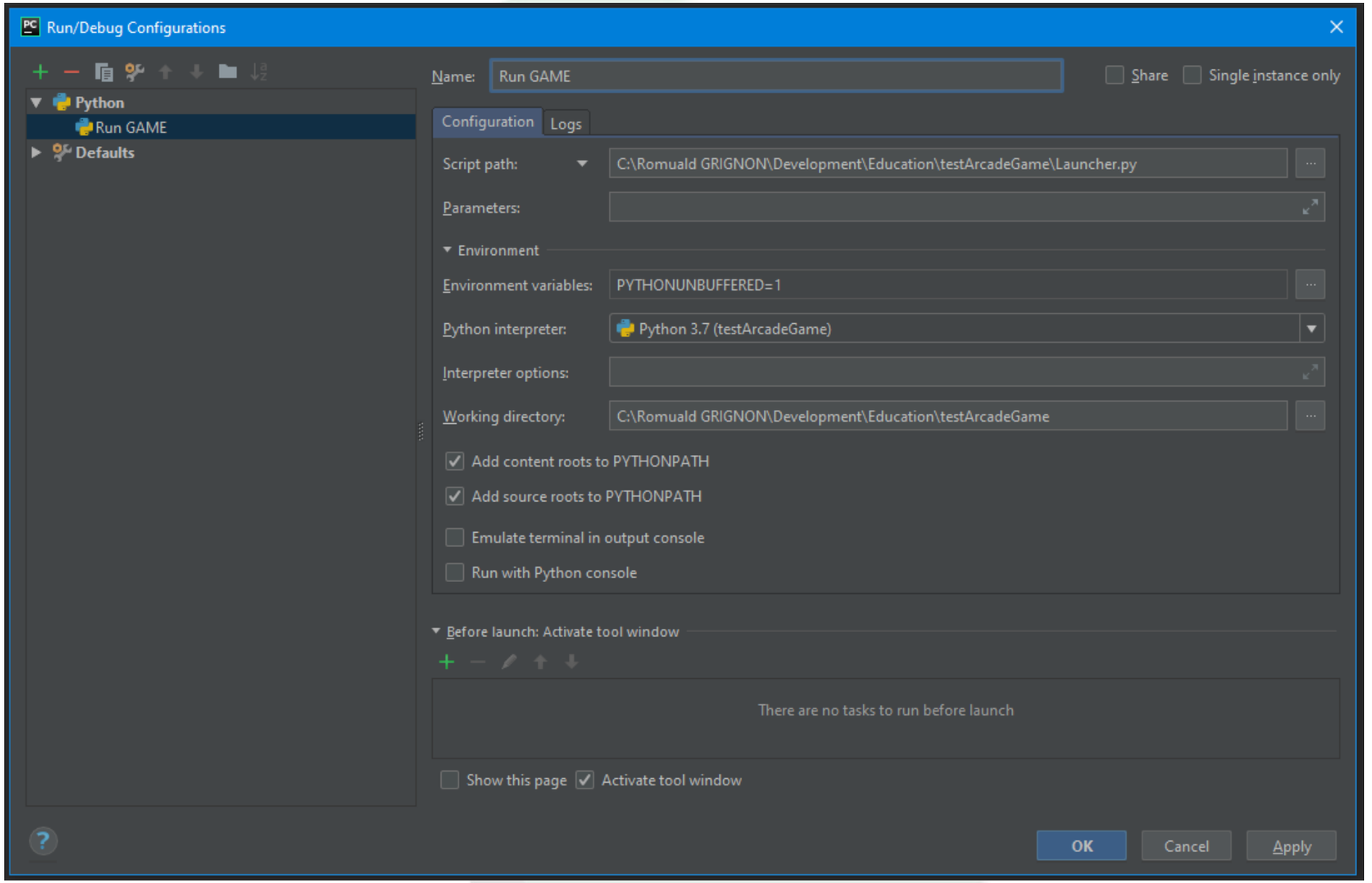
- Cliquez sur le bouton « + » vert pour créer une nouvelle configuration
- Sélectionnez le choix « Python » dans la liste déroulante qui apparaît



Environnement de travail

- Donnez un nom à votre configuration en écrivant dans le champ « Name »
- Sélectionnez le fichier Python principal, qui servira de point d'entrée du programme (ici il faut aller chercher le fichier « Launcher.py » dans le répertoire racine des fichiers sources qui vous ont été fournis)
- Vérifiez que l'interpréteur Python sélectionné (champ « Python Interpreter ») est bien celui qui a été créé dans votre sous-dossier venv
- Vérifiez également que le dossier de travail (champ « Working Directory ») est bien le dossier racine de votre projet
- Vous pouvez valider vos modifications
- (cf. capture d'écran de la fenêtre de configuration dans la diapositive suivante)

Environnement de travail



Environnement de travail

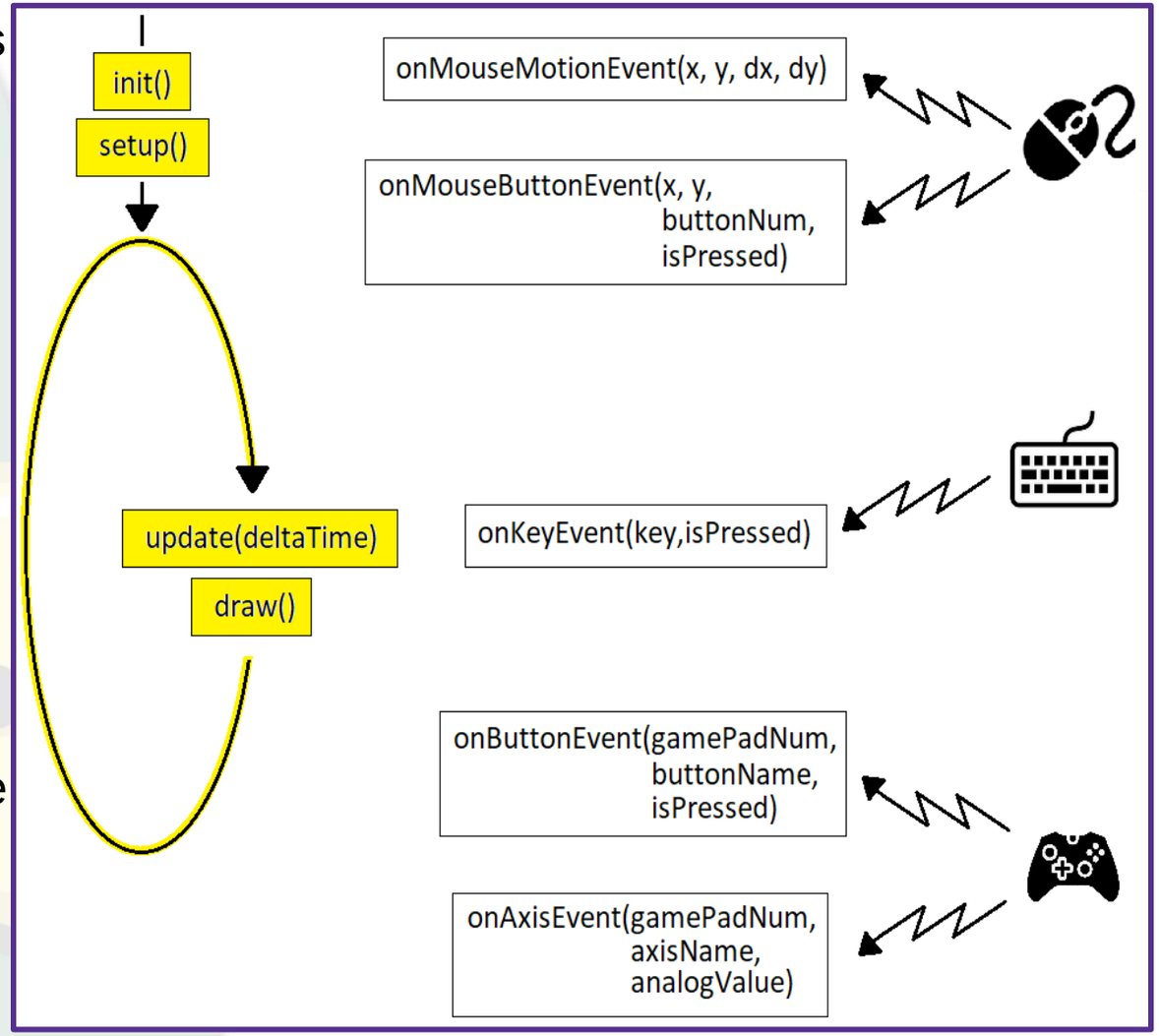
- Dans la fenêtre principale de PyCharm, vous avez maintenant votre configuration qui a été ajoutée à la liste déroulante en haut à droite
- Vous avez normalement un triangle vert à côté (symbole « play ») qui vous permet d'exécuter votre projet
- Cliquez dessus : vous devriez voir apparaître une fenêtre de jeu → Votre environnement de travail est prêt !!
- Si votre projet ne démarre pas, il peut s'agir d'incompatibilités avec des bibliothèques qu'utilise « arcade ». En effet cette dernière est toujours en développement
- Vérifiez bien que vous avez la version **2.4.3 de arcade** et la bibliothèque **pymunk** en version **5.7.0**. Vérifiez également que l'interpréteur python utilisé n'est pas la version 3.10
- Si des problèmes persistent, nous verrons ensemble comment corriger tout cela sur vos machines respectives

MOTEUR DE JEU

Moteur de jeu

- D'une manière générale, un moteur de jeu 2D possède la structure suivante

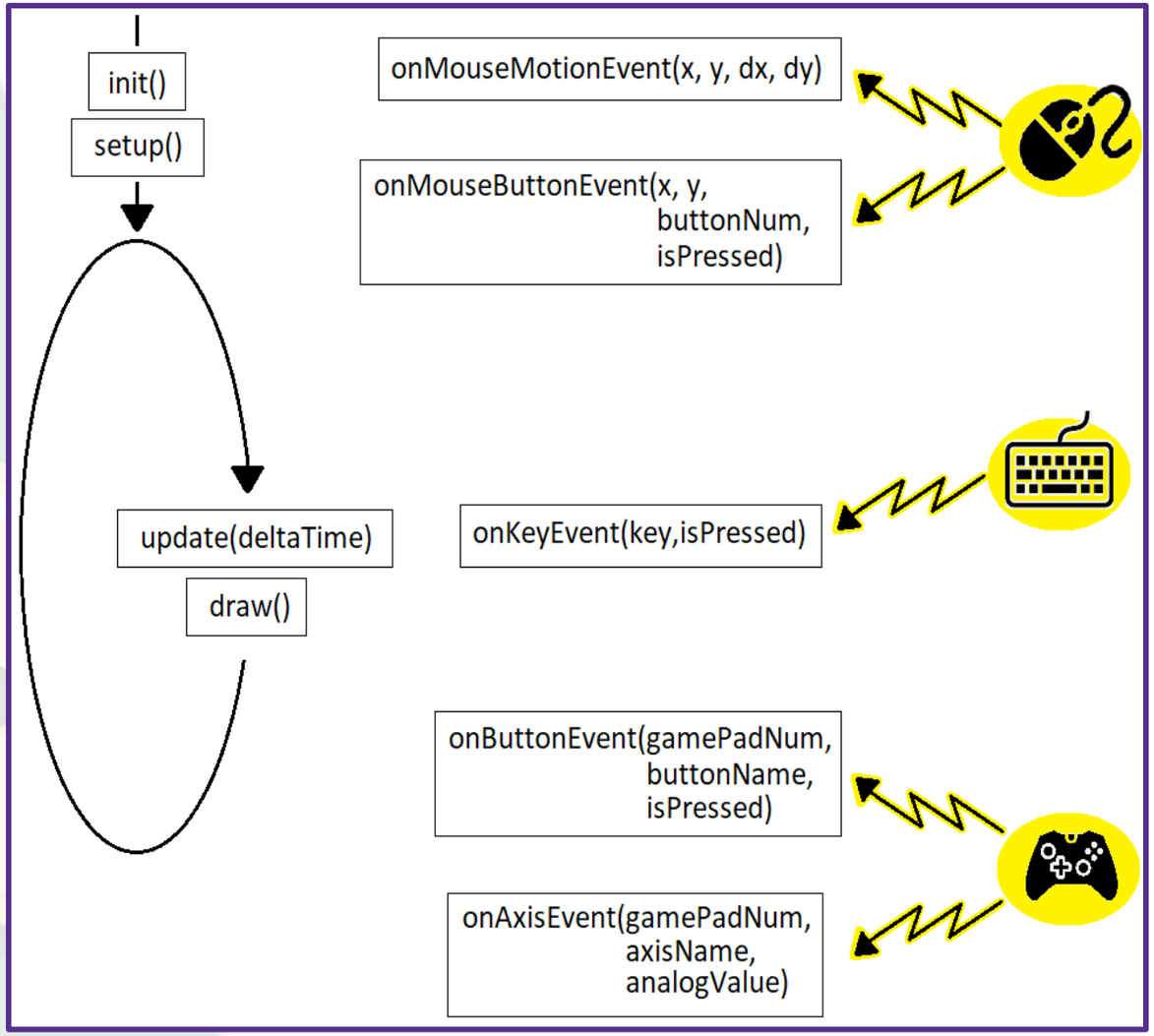
- Une phase d'initialisation du modèle de données (variables du jeu), ici centralisée dans les fonctions init() et setup()
- Une **boucle de jeu** (infinie) qui va, pour chaque image affichée à l'écran ('frame'), mettre à jour le modèle de données, dans la fonction update(), puis afficher les éléments graphiques à l'écran, via la fonction draw()
- La fonction **update()**, récupère en paramètre, une valeur réelle en secondes correspondant au temps depuis le dernier appel à cette même fonction. Ceci permet d'avoir une information sur le nombre de frames par seconde, et ainsi de modifier le modèle de données en ayant un référentiel de temps absolu (ceci est très utile dans les déplacements des éléments graphiques, qui ne doivent pas être impactés par les variations de rafraichissement de l'écran)
- La fonction **draw()** servira à afficher, ou non, explicitement certains éléments graphiques, ceci en fonction de la problématique du jeu



Moteur de jeu

- Il existe un mécanisme qui permet de récupérer l'état des périphériques

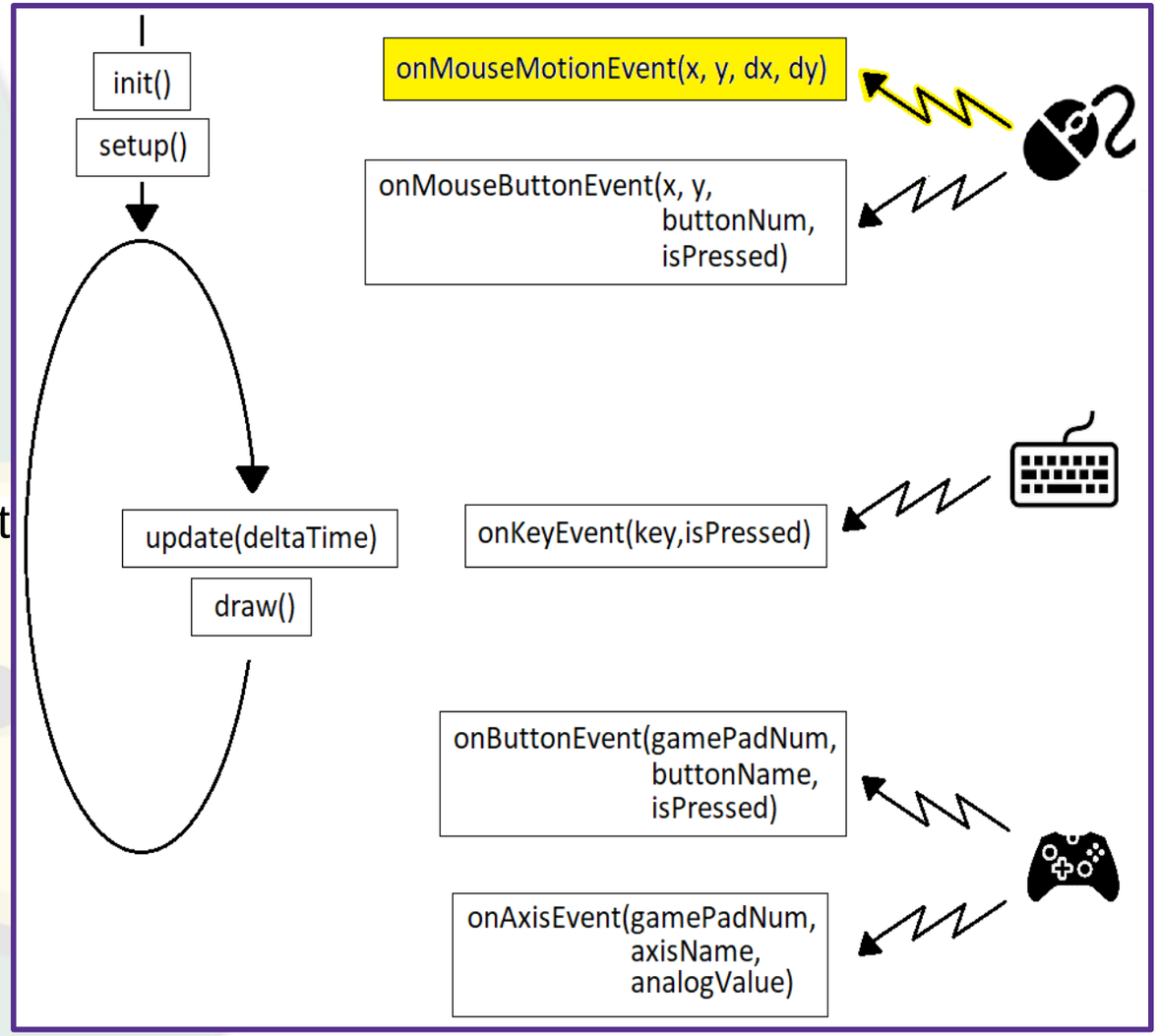
- La boucle de jeu est un traitement séquentiel qui ne permet pas de gérer de manière asynchrone les entrées du clavier ou des contrôleurs de jeu
- Pour cela, il existe en général un mécanisme basé sur des **interruptions**, qui permet d'interrompre l'exécution de la boucle de jeu, et exécuter une fonction qui récupérera l'**événement du périphérique** pour le traiter
- Nous avons ici 5 fonctions qui seront donc appelées de manière asynchrone par le moteur de jeu, en fonction des périphériques
- Ces fonctions permettent de capter les appuis sur les touches du **clavier**, le mouvement de la **souris**, les appuis sur les **boutons** des contrôleurs de jeu et de la souris, ainsi que les variations sur les **axes** (joysticks) de vos contrôleurs de jeu USB



Moteur de jeu

- **onMouseMotionEvent(x, y, dx, dy)** : déplacements de la souris

- **onMouseMotionEvent** est exécutée quand la souris est déplacée
- Les paramètres '*x*' et '*y*' contiennent les coordonnées de la souris à l'écran (en pixels)
- Les paramètres '*dx*' et '*dy*' contiennent la variation du mouvement de la souris (en pixels relatifs) par rapport à la position connue précédente. Ces variations sont donc positives ou négatives en fonction du sens de déplacement de la souris
- L'ensemble des périphériques enregistrés comme des pointeurs de souris impacterons cette fonction



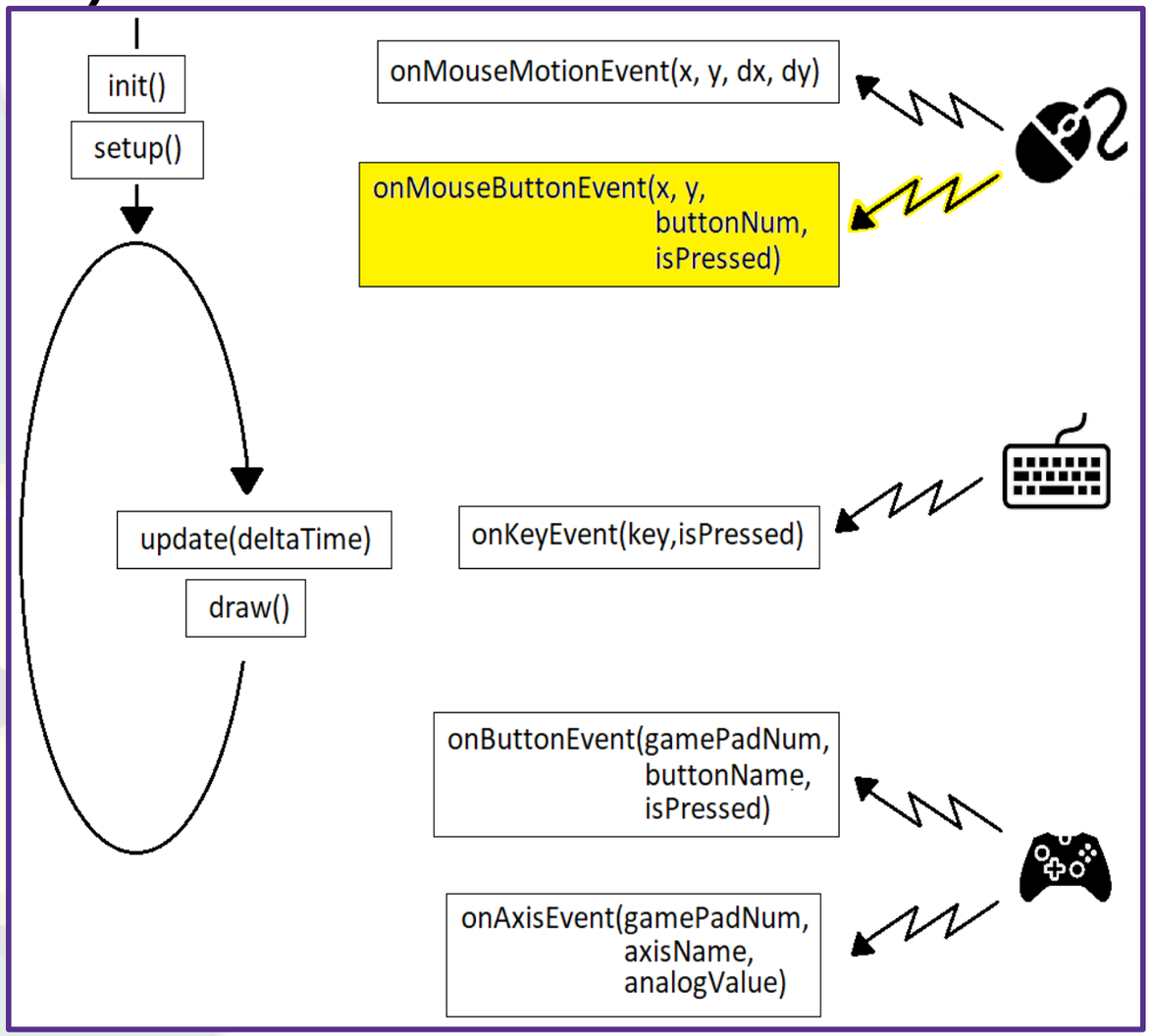
INFO : L'origine est située dans le coin inférieur/gauche de l'écran. L'axe X progresse vers la droite et l'axe Y progresse vers le haut

Moteur de jeu

- **onMouseButtonEvent(x, y, buttonNum, isPressed)** : boutons de la souris

- **onMouseButtonEvent** est exécutée quand un bouton de la souris est enfoncé ou relaché
- Les paramètres '*x*' et 'y' contiennent les coordonnées de la souris à l'écran (en pixels) au moment de l'événement
- Le paramètre '*buttonNum*' est une valeur entière qui indique quel est le numéro du bouton de la souris
- Le paramètre '*isPressed*' est un booléen qui contient **True/False** si le bouton est enfoncé/relaché respectivement
- L'ensemble des périphériques enregistrés comme des pointeurs de souris impacterons cette fonction

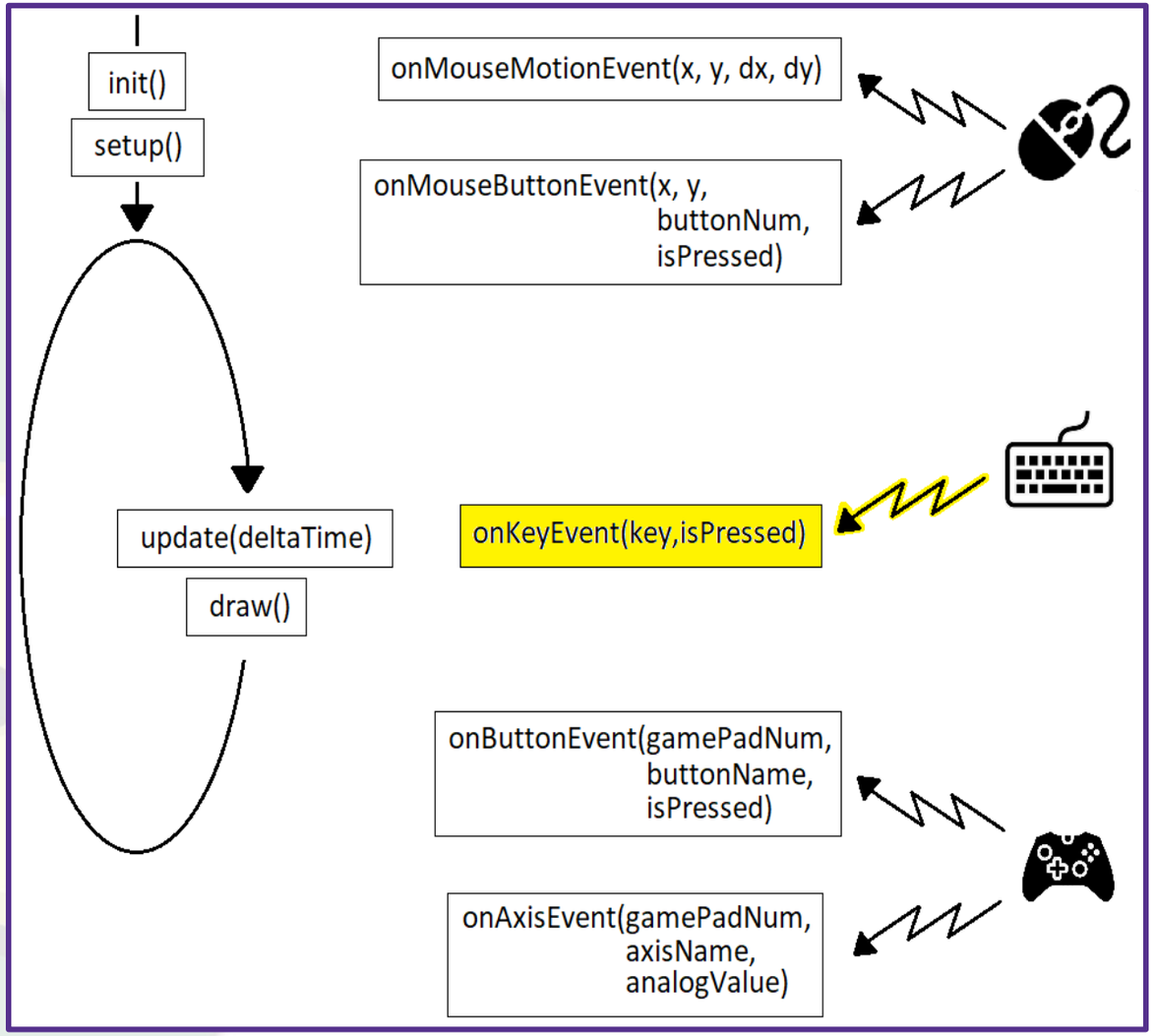
INFO : L'origine est située dans le coin inférieur/gauche de l'écran. L'axe X progresse vers la droite et l'axe Y progresse vers le haut



Moteur de jeu

- **onKeyEvent(key, isPressed)** : touches du clavier

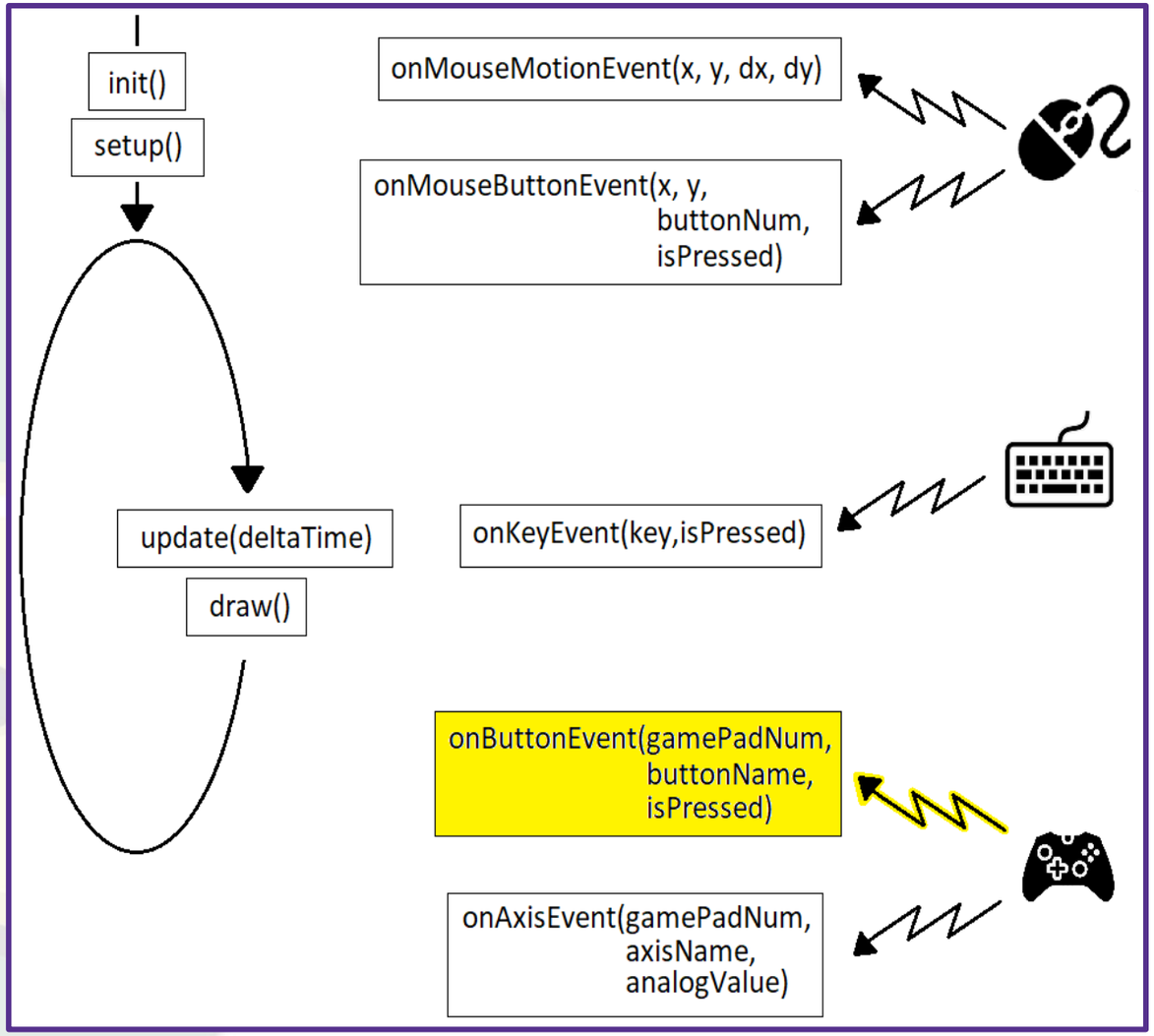
- **onkeyEvent** est exécutée quand une touche est enfoncée ou relachée
- Le paramètre '*key*' contient le code de la touche correspondante
- Le paramètre '*isPressed*' est un booléen qui contient **True/False** si la touche est enfoncée/relachée respectivement
- Pour tester la valeur de '*key*', il faut la comparer aux valeurs de '***arcade.key.xxxxx***'



Moteur de jeu

- **onButtonEvent(gamePadNum, buttonName, isPressed)** : boutons des contrôleurs

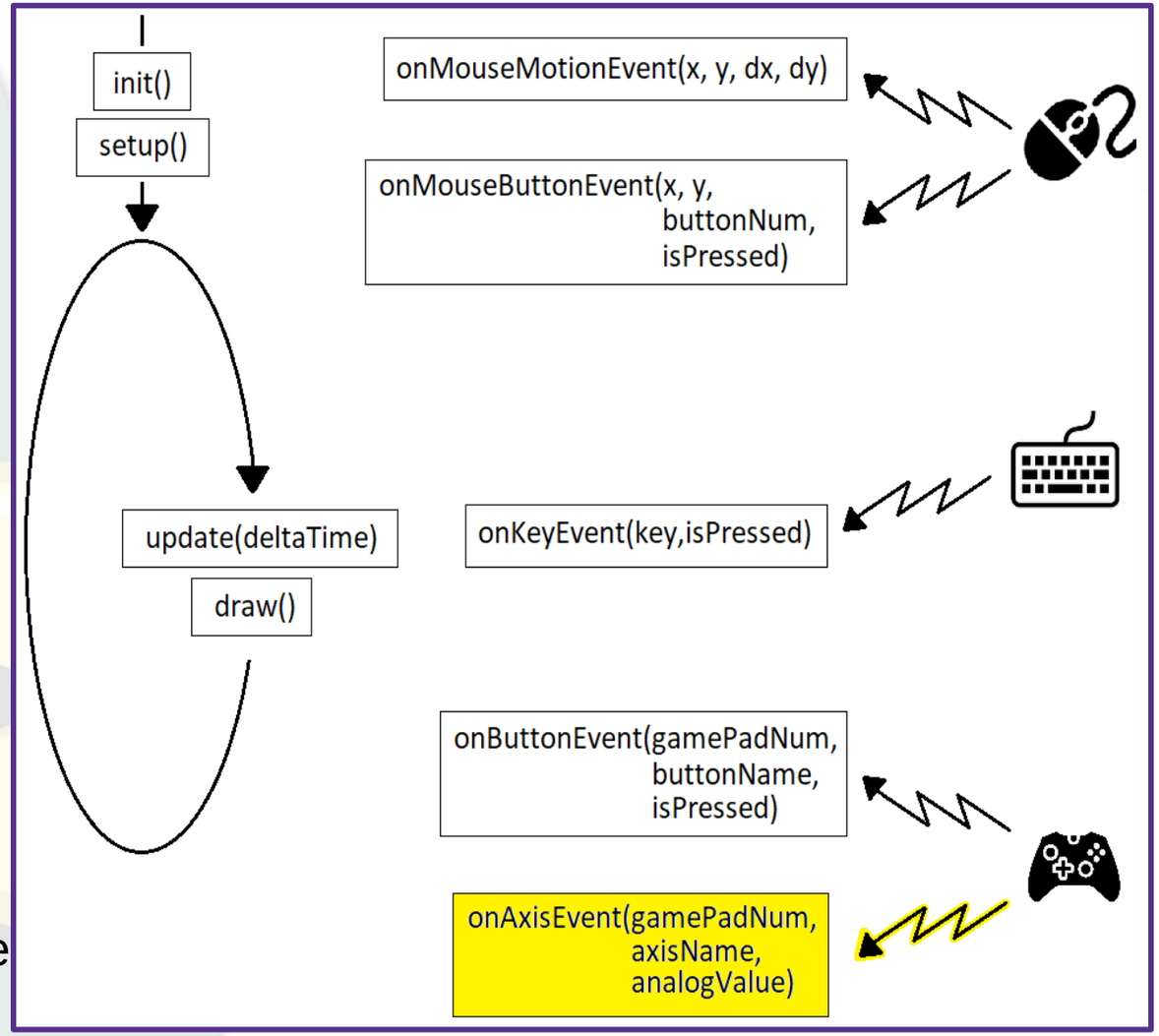
- **onButtonEvent** est exécutée quand un bouton de n'importe quel contrôleur USB est appuyé ou relâché
- Le paramètre '*gamePadNum*' contient le numéro du contrôleur à l'initiative de l'événement. Ceci permet de repérer les manettes entre elles dans le cas d'un jeu multi-joueurs local
- Le paramètre '*buttonName*' contient une chaîne de caractères avec le nom fonctionnel du bouton (ex : « **A** », « **B** », « **X** », « **Y** », « **LB** », « **RB** », « **LSTICK** », « **RSTICK** », « **VIEW** », « **MENU** »)
- Le paramètre '*isPressed*' est un booléen qui contient **True/False** si le bouton du gamepad est enfoncé/relâché respectivement



Moteur de jeu

- **onAxisEvent(gamePadNum, axisName, analogValue)** : axes des contrôleurs

- **onAxisEvent** est exécutée dès qu'un des axes d'un contrôleur USB est déplacé
- Le paramètre '*gamePadNum*' contient le numéro du contrôleur à l'initiative de l'événement. Ceci permet de repérer les manettes entre elles dans le cas d'un jeu multi-joueurs local
- Le paramètre 'axis*Name*' contient une chaîne de caractères avec le nom fonctionnel de l'axe (ex : « **X** », « **Y** », « **RX** », « **RY** », « **Z** »)
- Le paramètre '*analogValue*' est un réel qui contient la valeur de déplacement de l'axe, normalisée entre **-1.0** et **+1.0**
- Pour un axe horizontal, une valeur négative correspond à une inclinaison sur la gauche
- Pour un axe vertical, une valeur négative correspond à une inclinaison vers le bas



SPRITES

Sprites : Définition

- Un sprite, est une image stockée en mémoire, et destinée à être affichée à l'écran
- Ce type de donnée sert donc à stocker un élément graphique du jeu
- Lorsqu'une variable de type sprite sera créée, au même titre qu'un type structuré que vous avez pu aborder dans vos premiers cours d'informatique, elle contiendra la référence vers l'objet sprite créé
- A partir de cette référence, nous pouvons manipuler le sprite pour récupérer ses propriétés (taille, position, échelle, angle, …) ou modifier ses propriétés (déplacer, tourner, filtrer les couleurs, …)
- Il existe des sprites fixes, qui ne contiennent qu'une seule image constamment affichée, et des sprites animés, qui contiennent plusieurs images, appelées 'frames'. Les sprites animés permettront d'afficher successivement plusieurs images qui donneront l'illusion d'animation (comme le principe du dessin animé)
- Pour ce projet, des fonctions existent déjà pour initialiser une variable de type sprite en fonction de différents paramètres tels que le chemin de l'image à utiliser, la taille, la vitesse d'animation, … . Ces fonctions sont détaillées dans les diapositives qui suivent

createFixedSprite(...)

- Cette fonction retourne une référence vers un sprite créé à partir du dictionnaire passé en paramètre. Les champs de ce dictionnaire sont les suivants :
  - **filePath** : chaine de caractères contenant le chemin relatif de l'image à utiliser (ex : "images/items/star.png"). Ce champ est obligatoire.
  - **size** : tuple contenant deux valeurs qui sont, dans l'ordre, la largeur et la hauteur du sprite voulu (en pixels). Ces dimensions ne sont pas forcément celles de l'image d'original (le programme va automatiquement faire le redimensionnement). Ce champ est optionnel, et sa valeur par défaut contient la taille de l'image d'origine.
  - **filterColor** : un tuple contenant 4 valeurs, RGBA, permettant de filtrer les couleurs, et la transparence du sprite, pour différents effets graphiques. Ce champ est optionnel, et sa valeur par défaut est (255,255,255,255), ce qui ne modifie pas l'apparence de l'image d'origine.
  - **isMaxRatio** : un booléen qui permet d'indiquer le mode de redimensionnement. Si la valeur est *False*, alors le redimensionnement ne fera pas déborder l'image de la zone de taille *size*. Si la valeur est *True*, le redimensionnement fera en sorte que l'image remplisse complètement la zone de taille *size* (et donc peut être qu'une des dimensions débordera). Ce champ est optionnel, et sa valeur par défaut est *False*.
  - **position** : tuple contenant la position initiale sur l'écran de votre sprite. Ce tuple contient 2 valeurs qui sont, dans l'ordre, l'abscisse et l'ordonnée, en pixels. Ce champ est optionnel, et sa valeur par défaut est (0,0).

INFO : les coordonnées d'un élément graphique sont celles du centre de l'image

createAnimatedSprite(...)

- Cette fonction retourne une référence vers un sprite créé à partir du dictionnaire passé en paramètre. Les champs de ce dictionnaire sont ceux de la fonction createFixedSprite, et d'autres champs supplémentaires qui sont les suivants :
  - **spriteBox**: tuple de 4 valeurs qui permet d'indiquer combien de frames sont présentes dans l'image d'origine, à la fois sur l'axe X et sur l'axe Y, et quelle est la taille de chacune de ces frames (largeur et hauteur, en pixels). Exemple de valeur pour une image contenant 6 frames horizontales et 3 frames verticales, chacune de largeur 120px et de hauteur 75px : (6, 3, 120, 75). Ce champ est obligatoire.
  - **startIndex**: valeur entière contenant l'index de la première frame à afficher. L'index commence à 0, et pour chaque frame lue dans le sens de lecture (de gauche à droite puis de bas en haut), l'index est incrémenté. Dans l'exemple précédent, il y a 18 frames, de 0 à 17. Ce champ est obligatoire.
  - **endIndex** : valeur entière contenant l'index de la dernière frame à afficher, comme pour le champ startIndex. Ce champ est obligatoire.
  - **frameDuration** : un réel qui contient la durée d'affichage de chaque frame, en secondes. Ce champ est optionnel et sa valeur par défaut est de 1/60 secondes.
- Les 2 champs suivants sont communs aux Sprites fixes et animés :
  - **flipH**: booléen qui permet d'effectuer un miroir horizontal des images. Champ optionnel (défaut = *False)*.
  - **flipV**: booléen qui permet d'effectuer un miroir vertical des images. Champ optionnel (défaut = *False)*.

Manipulation des sprites

- Lire les propriétés :
  - print(mySprite.**center_x**) ## affiche la position X du centre du sprite en pixels
  - print(mySprite.**center_y**) ## affiche la position Y du centre du sprite en pixels
  - print(mySprite.**width**) ## affiche la largeur du sprite en pixels
  - print(mySprite.**height**) ## affiche la hauteur du sprite en pixels
  - print(mySprite.**angle**) ## affiche l'angle du sprite en degrés
- Modifier la position du centre :
  - *mySprite*.**center_x** = 500 ## place le centre du sprite à 500 pixels du bord gauche de l'écran
  - *mySprite*.**center_y** = 275 ## place le centre du sprite à 275 pixels du bord bas de l'écran
- Modifier l'angle :
  - *mySprite*.**angle** += 15 ## faire tourner le sprite de 15 degrés dans le sens horaire
- Afficher le sprite à l'écran :
  - *mySprite*.**draw()** ## affiche le sprite comme configuré précédemment

Manipulation spécifique des sprites animés

- Mettre a jour les images automatiquement :
  - *mySprite*.**update_animation(...)** ## calcule automatiquement quelle image doit être sélectionnée en fonction de la valeur « frameDuration » et du nombre d'images
    ## Peut prendre un paramètre optionnel qui est le temps écoulé depuis la dernière image
- Sélectionner l'image manuellement :
  - *mySprite*.**set_texture(idx)** ## sélectionne manuellement l'image numéro idx (cette valeur doit être cohérente avec le nombre d'images, et commence à 0)
  - *Print(mySprite*.**cur_texture_index)** ## affiche le numéro de l'image actuellement sélectionnée
- Pour afficher le sprite, on utilisera bien sûr la fonction draw(), comme indiqué dans la diapositive précédente, qui se chargera d'afficher l'image choisie (soit automatiquement, soit manuellement)

EFFETS de PARTICULES

Effets de particules : présentation

- Un effet de particules est un mécanisme permettant d'afficher de nombreux éléments graphiques (particules) ayant tous un comportement pseudo-aléatoire (mouvement, orientation, couleur, ...), comme des étincelles par exemple.
- Le système d'effets de particule utilisé ici va envoyer les particules dans toutes les directions, et ce de manière aléatoire
- Chaque particule va donc avoir sa propre position, orientation, vitesse, couleur, transparence, durée de vie, …, et le système d'effets de particules va s'occuper de mettre a jour chacune de ces propriétés pour ensuite pouvoir les afficher toutes à l'écran
- Dans la sémantique utilisée, vous pourrez également voir le terme : 'émetteur' de particules
- De notre point de vue, un système d'effets de particules sera simplement une référence stockée dans une variable, à partir de laquelle nous pourrons appeler des fonctions pour le mettre à jour et l'afficher.
- Dans les diapositives suivantes, nous allons détailler les fonctions qui permettent de créer des variables de type 'effets de particules', et les diverses fonctions qui permettent de les manipuler

Effets de particules : types

- Il existe deux fonctions qui permettent de créer des effets de particules qui ont chacun un comportement différent :
  - **createParticleEmitter(...)** : effet de particules permanent. Ce type d'effet de particules est destiné à être toujours actif. Il servira à créer des effets graphiques constamment affichés à l'écran comme la flamme d'un réacteur ou d'une torche par exemple.
  - **createParticleBurst(...)** : effet de particules de type 'explosion'. Ce type d'effet sera temporaire, souvent très bref. Il permet d'ajouter de la dynamique graphique à un moment très précis du jeu. Par exemple générer des étincelles quand une voiture cogne contre un obstacle, ou afficher des étoiles quand votre personnage récupère un bonus du jeu.
- Ces 2 fonctions renvoient une référence vers un objet de type 'effet de particules' :
  - myEmitter1 = createParticleEmitter(...)
  - myEmitter2 = createParticleBurst(...)
- Pour les 2 types d'effets, il existe des paramètres en commun qui seront détaillés plus loin dans ce document. Il existe également des paramètres spécifiques, détaillés eux aussi dans des diapositives distinctes.

Effets de particules : création (commune)

- Pour les deux fonctions de création d'effets de particules, *createParticleEmitter(...)* et *createParticleBurst(...)*, le seul paramètre est un dictionnaire. Les champs communs de ces deux fonctions sont les suivants :
  - **position** : tuple (X,Y) de la position de référence de l'émetteur (valeur numérique en pixels)
  - **filePath** : comme pour les Sprites, le chemin du fichier contenant les images
  - **spriteBox** : comme pour les Sprites, un tuple indiquant le format des images dans le fichier « filePath »
  - **spriteSelect** : un tuple (x, y) indiquant l'image sélectionnée. Les indices commencent à 0
  - **partSize** : taille de chaque particule (en pixels), en partant de l'hypothèse qu'elle est carrée ou circulaire.
  - **partScale** : échelle de redimensionnement de chaque particule (valeur réelle). Pour un meilleur rendu, il est préférable de maximiser le champ *partSize* et réduire la taille de la particule avec le champ *partScale*.
  - **partSpeed** : valeur réelle indiquant la vitesse de déplacement de chaque particule
  - **filterColor** : tuple de 4 valeurs entières (RGBA) pour fixer la couleur de chaque particule
  - **startAlpha** : valeur entière entre 0 et 100 indiquant la transparence en début de vie de la particule
  - **endAlpha** : valeur de transparence en fin de vie de la particule. (0 = invisible / 100 = opaque)

Effets de particules : création (émetteur infini)

- Pour la fonction createParticleEmitter(...), les champs spécifiques du dictionnaire passé en paramètres sont les suivants :
  - **partNB :** valeur entière pour le nombre maximal de particules de l'émetteur. Ce champ est obligatoire.
  - **maxLifeTime :** valeur réelle indiquant le temps pendant lequel chaque particule va être affichée à l'écran. Les particules auront donc une durée de vie unitaire comprise entre « maxLifeTime/10 » et « maxLifeTime ». Ce champ est obligatoire.

- Ces deux champs permettent de configurer combien de particules simultanées seront affichées en même temps (au maximum) et combien de temps chacune restera affichée.
- En complément avec les champs précédents « startAlpha », « endAlpha », ou encore « partSpeed » , le rendu visuel sera différent : n'hésitez pas à tester plusieurs combinaisons.

Effets de particules : création (émetteur ‘Burst’)

- Pour la fonction createParticleBurst(...), les champs spécifiques du dictionnaire passé en paramètres sont les suivants :
  - **partInterval :** valeur réelle indiquant l'intervalle de temps entre 2 particules émises. Champ obligatoire.
  - **totalDuration :** valeur réelle indiquant le temps total de génération des particules. Champ obligatoire.

- Ces deux champs permettent d’indiquer un temps total de fonctionnement de l’émetteur. A l’inverse de l’émetteur précédent, le « Burst » ne dure pas indéfiniment, mais s’éteint au bout d’un temps compris entre « totalDuration/4 » et « totalDuration ».
- Tout comme le précédent émetteur, testez plusieurs combinaisons de paramètres pour trouver le rendu visuel adéquat.

Effets de particules : utilisation

- A partir de variables d'effets de particules (continus ou bursts), il existe des fonctions et des propriétés permettant de les manipuler :
  - *myEmitter*.**center_x** = 500 ## déplace le centre de l'émetteur au pixel 500 sur l'axe X
  - *myEmitter*.**center_y** = 70 ## déplace le centre de l'émetteur au pixel 70 sur l'axe Y
  - *myEmitter*.**update()** ## met à jour les propriétés des particules de l'émetteur
  - *myEmitter*.**draw()** ## affiche toutes les particules de l'émetteur à l'écran

- Pour les émetteurs de particules de type 'burst' , il est possible de savoir quand l'émetteur a terminé de générer les particules, dans le but de le supprimer de la mémoire :
  - *myEmitter2*.**can_reap()** ## fonction qui retourne un booléen qui indique si l'émetteur a terminé sa génération de particules et si il peut être désalloué de la mémoire

COLLISIONS

Collisions:

- Une fonctionnalité classique dans les jeux vidéo est de détecter si des éléments sont en contact/superposition
- Que ce soit pour déterminer si un personage est touché par un projectile, si il a réussi à attraper un objet bonus, ou pour bloquer un mouvement, la détection de collision est incontournable
- Dans le code source qui vous est fourni, vous disposez déjà de fonctions permettant de faire de la détection de formes simples
- Ces fonctions permettent une détection de 2 formes parmi les suivantes :
  - Des cercles
  - Des rectangles alignés sur les axes du repère (AABB ou Axis-Aligned Bounding Box)
  - Des ellipses avec ses axes alignés sur ceux du repère

collision2Circles (...)

- Cette fonction retourne un booléen si les deux formes sont en collision :
  - **center1** : position du centre (tuple) du premier cercle en pixels
  - **radius1** : rayon du premier cercle en pixels
  - **center2** : position du centre (tuple) du deuxième cercle en pixels
  - **radius2** : rayon du deuxième cercle en pixels

collision2AABB (...)

- Cette fonction retourne un booléen si les deux formes sont en collision :
  - **topLeft1** : position du coin en haut à gauche (tuple) du premier rectangle en pixels
  - **bottomRight1** : position du coin en bas à droite (tuple) du premier rectangle en pixels
  - **topLeft2** : position du coin en haut à gauche (tuple) du deuxième rectangle en pixels
  - **bottomRight2** :position du coin en bas à droite (tuple) du premier rectangle en pixels

collisionCircleAABB (...)

- Cette fonction retourne un booléen si les deux formes sont en collision :
  - **topLeft** : position du coin en haut à gauche (tuple) du rectangle en pixels
  - **bottomRight** : position du coin en bas à droite (tuple) du rectangle en pixels
  - **center** : position du centre (tuple) du cercle en pixels
  - **radius** : rayon du cercle en pixels

collisionCircleEllipse (...)

- Cette fonction retourne un booléen si les deux formes sont en collision :
  - **center1** : position du centre (tuple) du cercle en pixels
  - **radius1** : rayon du cercle en pixels
  - **center2** : position du centre (tuple) de l'ellipse en pixels
  - **radiusX2** : demi-axe X de l'ellipse en pixels
  - **radiusY2** : demi-axe Y de l'ellipse en pixels

collision2Ellipses (...)

- Cette fonction retourne un booléen si les deux formes sont en collision :
  - **center1** : position du centre (tuple) de l'ellipse #1 en pixels
  - **radiusX1** : demi-axe X de l'ellipse #1 en pixels
  - **radiusY1** : demi-axe Y de l'ellipse #1 en pixels
  - **center2** : position du centre (tuple) de l'ellipse #2 en pixels
  - **radiusX2** : demi-axe X de l'ellipse #2 en pixels
  - **radiusY2** : demi-axe Y de l'ellipse #2 en pixels

collisionEllipseAABB (...)

- Cette fonction retourne un booléen si les deux formes sont en collision :
  - **topLeft** : position du coin en haut à gauche (tuple) du rectangle en pixels
  - **bottomRight** : position du coin en bas à droite (tuple) du rectangle en pixels
  - **center** : position du centre (tuple) de l'ellipse en pixels
  - **radiusX** : demi-axe X de l'ellipse en pixels
  - **radiusY** : demi-axe Y de l'ellipse en pixels

TEXTES BRUTS

drawText(...)

- Cette fonction affiche un texte à l'écran à partir du dictionnaire passé en paramètre (ATTENTION, cette fonction n’est pas optimisée et peut affecter les performances du jeu) :
  - **x** : valeur entière pour indiquer la position du texte sur l'axe X. Ce champ est obligatoire.
  - **y** : valeur entière pour indiquer la position du texte sur l'axe Y. Ce champ est obligatoire.
  - **message** : chaîne de caractères à afficher sur l'écran. Ce champ est obligatoire.
  - **size** : taille de la police de caractères (en points). Ce champ est optionnel et la valeur par défaut est 12.
  - **color** : un tuple contenant 4 valeurs, RGBA, pour fixer la couleur et la transparence du texte. Ce champ est optionnel, et sa valeur par défaut est (255,255,255,255).
  - **alignH** : chaîne de caractères indiquant l'alignement horizontal par rapport à la position de référence. Les valeurs possibles sont "left", "center", "right". Ce champ est optionnel, et sa valeur par défaut est "center".
  - **alignV** : chaîne de caractères indiquant l'alignement vertical par rapport à la position de référence. Les valeurs possibles sont "top", "center", "bottom". Ce champ est optionnel, et sa valeur par défaut est "center".
  - **angle** : valeur numérique indiquant l'orientation du texte en degrés. Ce champ est optionnel et sa valeur par défaut est 0.
  - **bold** : booléen qui indique si le texte est en gras. Ce champ est optionnel. Sa valeur par défaut est *False*
  - **italic** : booléen qui indique si le texte est en italique. Ce champ est optionnel. Sa valeur par défaut est *False*

SONS

Création et utilisation de sons

- Les objets de type son, sont des espaces mémoires chargés dans la RAM de l'ordinateur et qui sont utilisables pour les jouer à la demande.
- La fonction createSound(...) va créer et retourner la référence vers un objet son. Le seul paramètre est :
  - **fileName** : chaîne de caractères pour indiquer le chemin où se trouve le fichier son (ex : "sounds/bleep.wav")

- Pour jouer un son à partir de la variable, il suffit d'appeler la fonction play() :
  - *mySound*.**play()** ## joue le son sur les hauts parleurs de votre machine. Un paramètre optionnel pour gérer le volume est disponible. Il s'agit d'un nombre réel avec une valeur par défaut de 1.0

DIVERS

Touches spécifiques:

- L’application peut être quittée en pressant la touche **ESC** (si le focus est sur la fenêtre au moment de l’appui sur la touche bien sûr)

- La touche **F11** permet de basculer entre l’affichage fenêtré par défaut, et l’affichage plein écran. Si le ration Largeur/Hauteur de la fenêtre ne respecte pas celui de l’écran, la scène sera automatiquement centrée sur l’écran.