Pygame

Introduction à Pygame

Traduit de l'anglais, l'original par *Pete Shinners* :
https://www.pygame.org/docs/tut/PygameIntro.html

Introduction à Pygame

Cet article est une introduction à la bibliothèque Pygame pour les programmeurs Python. La version originale est parue dans le Pyzine volume 1 issue 3. Cette version contient des révisions trop mineures pour créer un nouvel article. Pygame est une bibliothèque d'extension de Python (https://www.python.org), enveloppe de la bibliothèque SDL (http://www.libsdl.org).

Histoire

Pygame a commencé durant l'été 2000. Connaissant le langage C depuis des années, j'ai découvert Python et SDL en même temps. Si vous êtes déjà familiarisé avec Python, (qui en était à la version 1.5.2), vous pourriez avoir besoin d'une introduction à SDL : *Simple Directmedia Librairy*. Créée par Sam Lantinga, SDL est une bibliothèque multi-plateforme écrite en C afin de gérer le multimédia, elle est comparable à DirectX. Elle a été utilisée par des centaines de projets commerciaux et de jeux open-source. J'ai été impressionné par la propreté et la simplicité des deux projets et j'ai vite réalisé qu'associer Python et SDL était une idée intéressante.

J'ai découvert un petit projet existant avec exactement la même idée : PySDL. Créé par Mark Baker, PySDL était une implémentation de SDL pour Python. L'interface était propre mais je trouvais que ce code était trop proche du C. La mort de PySDL m'a encouragé à lancer un nouveau projet.

J'ai cherché à faire un projet tirant réellement profit de Python. Mon but était de permettre de réaliser facilement des choses simples aussi bien que des choses plus complexes. Pygame a commencé en octobre 2000. Six mois plus tard, la version 1.0 faisait son apparition.

Un avant-goût

Je trouve que le meilleur moyen pour saisir le fonctionnement d'une nouvelle bibliothèque est de directement regarder un exemple parlant. Dans les premiers jours avec Pygame, j'ai créé une animation de balle rebondissante en 7 lignes de code. Jetons un coup d'œil à une version simplifiée de la même chose.

```
1  import sys, time, pygame
2  pygame.init()
3
4  size = width, height = 320, 240
5  speed = [2, 2]
```

```
 6 black = 0, 0, 0
 7
 8 screen = pygame.display.set_mode(size)
 9
10 ball = pygame.image.load("intro_ball.gif")
11 ballrect = ball.get_rect()
12
13 while 1:
14     for event in pygame.event.get():
15         if event.type == pygame.QUIT: sys.exit()
16
17     ballrect = ballrect.move(speed)
18     if ballrect.left < 0 or ballrect.right > width:
19         speed[0] = -speed[0]
20     if ballrect.top < 0 or ballrect.bottom > height:
21         speed[1] = -speed[1]
22
23     screen.fill(black)
24     screen.blit(ball, ballrect)
25     time.sleep(0.01)
26     pygame.display.flip()
```

intro_ball.gif

(Si vous voulez essayer cet exemple, mettez une image nommée *intro_ball.gif* dans le même dossier.)

- C'est la façon la plus simple d'obtenir une animation de balle rebondissante. On voit au début du code l'importation et l'initialisation du module Pygame, ce qui ne change pas vraiment d'un programme à un autre. Le `import pygame` importe toutes les modules disponibles de Pygame. L'appel de `pygame.init()` initialise chacun de ces modules.

- *À la ligne 8*, une fenêtre graphique est créée par l'appel de `pygame.display.set_mode()`. Pygame et SDL nous simplifie la vie en sélectionnant par défaut les modes graphiques les plus adaptés au matériel graphique. Vous pouvez outrepasser ce mode et Pygame compensera tout ce que le matériel ne peut pas faire. Pygame représente les images comme des objets Surface. La fonction `display.set_mode()` crée un nouvel objet Surface représentant le graphisme actuel à afficher. Tous les dessins que vous faites sur cette surface seront visibles à l'écran.

- *À la ligne 10*, nous chargeons notre image de balle. Pygame supporte une grande variété de format d'image, comme la bibliothèque *SDL_image*, les formats pris en compte sont : JPG, PNG, TGA et GIF. La fonction `pygame.image.load()` nous retourne la surface contenant l'image de balle. La surface gardera toutes les propriétés (couleur clé et/ou de transparence alpha) du fichier d'origine. Après le chargement de l'image de la balle, nous créons une variable `ballrect`. Pygame possède un objet très utile nommé `Rect` qui représente une zone rectangulaire. Dans la partie du code chargée de l'animation, nous verrons quel est le rôle des objets `Rect`.

- *À la ligne 13*, notre programme est initialisé et prêt à être lancé. À l'intérieur d'une boucle infinie qui gère les évènements générés par l'utilisateur, le code dessine et fait bouger la balle. Si vous êtes familier avec la programmation de GUI (*Graphical User Interface*, ou Interface Utilisateur Graphique), vous avez certainement une notion de ce que sont les

évènements. Avec Pygame, il s'agit de la même chose. Lorsqu'un évènement se produit, le programme l'intercepte et agit en conséquence. Le premier évènement pris en compte est l'action de quitter le programme. Si cet évènement survient, le programme agit en conséquence en appelant la méthode `sys.exit()`.

- Ensuite vient le traitement de l'actualisation de la position de la balle. *Les lignes 17 et 21* déplacent la variable `ballrect` par sa vitesse courante. Si la balle s'est déplacée à l'extérieur de l'écran, la vitesse de la balle est inversée pour qu'elle reparte dans l'autre sens. Ça ne correspond pas exactement aux lois de Newton, mais ce n'était pas l'objectif de ce premier programme de toute façon.

- *À la ligne 23*, nous effaçons l'écran en le remplissant de noir. Si vous n'avez jamais travaillé avec des animations, ceci peut vous paraître étrange. Vous pouvez vous demander : *Pourquoi avons-nous besoin de tout effacer ? Pourquoi n'avons-nous pas juste la balle à bouger sur l'écran ?*, ceci n'est pas vraiment la manière de fonctionner des animations sur ordinateur. Une animation n'est rien de plus qu'une succession d'images, mais cette succession des images est tellement rapide que le cerveau humain voit une animation. `screen` n'est qu'une image que l'utilisateur voit. Si vous n'avez pas pris le temps d'effacer la balle de l'écran, nous pourrions voir la trace de la balle tout en continuant de voir la balle dans ses nouvelles positions.

- *À la ligne 24*, nous dessinons l'image de la balle sur l'écran. Le dessin de ces images est géré par la méthode `Surface.blit()`. Un *blit* signifie que l'on copie les couleurs de chaque pixel d'une image sur une autre. Avec la méthode `blit()`, on prend une surface dite *source* que l'on applique sur la surface *destination*, le tout à une position définie.

- *À la ligne 25*, le programme attend 10 millisecondes entre chaque image afin que l'animation ne soit pas trop rapide.

- La dernière chose dont nous avons besoin, est la mise à jour de l'affichage visible. Pygame gère l'affichage avec un double tampon (double buffer). Quand nous avons fini de dessiner, nous faisons appel à la fonction `pygame.display.flip()`. Ceci fait que tout ce que nous avons dessiné sur la Surface `screen` devient visible. Cette mise en tampon fait que nous sommes sûr de voir des images complètes dessinée à l'écran. Sans cette manipulation par tampon, l'utilisateur verrait l'image se dessiner au fur et à mesure sur son écran.

Ceci conclut notre courte introduction à Pygame. Pygame possède plusieurs modules pour gérer les entrées de la souris, du clavier et du joystick. Il est aussi possible de mixer des fichiers audio et de décoder des flux musicaux. Avec les Surfaces, vous pouvez dessiner de simples formes (ronds, ...), les faire pivoter, et/ou les mettre à une certaine échelle. Avec en plus un module pour des manipulations de pixels sur les images en temps réel avec les *NumPy arrays*. Pygame peut lire les videos MPEG et supporte les CDs audio. Pygame a également la capacité d'agir en tant que couche d'affichage multi-plateforme pour PyOpenGL. La plupart des modules de Pygame sont écrits en C, peu sont en Python.

Le site internet de Pygame possède une documentation de référence complète pour chaque fonction de Pygame et plusieurs tutoriels pour tout niveau d'utilisateur. Les sources de Pygame viennent avec de nombreux exemples pour une compréhension facilitée.

Python et les jeux

Python est il approprié pour les jeux ?
La réponse est : *Cela dépend du jeu.*

Python est en fait capable de faire tourner des jeux. Cela vous surprendra toujours de savoir tout ce qu'il est possible de faire en 30 millisecondes. Toutefois, il n'est pas difficile de plafonner une fois que votre jeu commencera à devenir plus complexe. N'importe quel jeu fonctionnant en temps réel utilisera pleinement l'ordinateur.

Depuis plusieurs années, il y a une tendance intéressante dans le développement de jeux : l'utilisation de langage de plus haut niveau (plus proches de l'utilisateur que de la machine). Généralement un jeu est divisé en 2 parties majeures. Le moteur de jeu, qui doit être le plus rapide possible, et la logique de jeu, qui commande le moteur de jeu. Il n'y a pas si longtemps, le moteur était écrit en assembleur avec quelques portions de C. De nos jours, le C est plus présent dans les moteurs de jeu, tandis que le jeu en lui même est écrit en langage de script de haut niveau. Les jeux comme *Quake3* et *Unreal Tournament* exécutent ces scripts en tant que bytecode portable.

Dans le courant 2001, les développeurs de *Rebel Act Studios* ont fini leur jeu, *Severance: Blade of Darkness*. Ce jeu utilise un moteur de jeu 3D modifié, le reste du jeu est écrit en Python. Le jeu est un jeu d'action à la 3ème personne. Vous contrôlez un guerrier médiéval attaquant et explorant des donjons. Vous pouvez télécharger le 3ème *add-on* pour ce jeu, et vous verrez qu'il n'y a rien d'autre que des fichiers sources en Python.

Beaucoup plus récemment, Python a été utilisé par une grande variété de jeux comme *Freedom Force* et *Humungous' Backyard Sports Series*.

Pygame et SDL s'utilisent comme un excellent moteur de jeu en C pour des jeux en 2D. Les jeux trouvent en grande partie ce dont ils ont besoin dans la SDL pour le graphisme. SDL peut avantageusement utiliser l'accélération graphique matérielle. Vous pouvez optimiser le jeu pour lui faire afficher entre 40 et 200 FPS (images par seconde). Lorsque l'on voit qu'un jeu en Python peut afficher 200 FPS, on réalise que Python et les jeux peuvent se combiner.

Il est impressionnant de savoir que Python et SDL fonctionne sur de multiple plateformes. Par exemple, en mai 2001, j'ai actualisé complètement mon projet avec Pygame : *SolarWolf*, un jeu d'action et d'arcade. Une chose qui m'a étonné est qu'un an après, il n'avait toujours pas eu besoin de patches, correction de bugs, ou de mises à jour. Le jeu était entièrement développé sous Windows mais tournait aussi sous Linux, Mac OSX et plusieurs Unix sans aucun travail de ma part.

Mais il y a clairement beaucoup de limitations. La meilleure manière de gérer l'accélération matérielle n'est pas toujours la meilleure façon d'avoir de meilleurs résultats pour une accélération logicielle. L'accélération n'est pas disponible sur toutes les plateformes. Quand un jeu devient trop complexe, il doit souvent se réduire à une seule plateforme. La SDL a également quelques limitations de conception, des choses comme le scrolling en plein écran peuvent réduire fortement la vitesse du jeu, jusqu'à ce que ça devienne injouable. SDL n'est pas fait pour tous les types de jeux, mais souvenez-vous que des compagnies comme Loki ont utilisé la SDL pour faire fonctionner une grande variété de jeux.

Pygame est un langage bas niveau quand il est utilisé pour coder les jeux. Vous aurez rapidement besoin d'envelopper des fonctions communes pour votre environnement de jeu. C'est en grande partie dû au fait qu'il n'y a rien dans Pygame pour ça. Votre programme a le contrôle total sur tout. Vous constaterez que l'effet indésirable à cela est que vous devrez écrire beaucoup de code ennuyeux pour obtenir un cadre plus avancé. Vous aurez besoin d'une meilleure compréhension de ce que vous faites.

En conclusion

Le développement en vaut la peine, il y a quelque chose d'excitant dans le fait de pouvoir voir et interagir avec le code que vous avez écrit. Pygame est actuellement utilisée par plus de 30 projets. Plusieurs sont jouables dès maintenant. Vous allez être surpris en visitant le site internet de Pygame (http://www.pygame.org) en voyant ce que les autres utilisateurs sont capables de faire avec Python.

Une chose qui a retenu mon attention, est que beaucoup de personnes se sont mises à utiliser Python pour leurs premiers essais en développement de jeu. Je peux voir pourquoi les jeux attirent de nouveaux programmeurs, mais il peut être difficile de créer un jeu qui requiert une compréhension plus solide du langage. J'ai essayé de soutenir ce type d'utilisateurs en écrivant plusieurs exemples et tutoriels sur Pygame pour des personnes nouvelles à ces concepts.

Pour finir, mon conseil est de faire au plus simple. Je ne peux me soumettre à une contrainte. Si vous prévoyez de créer votre premier jeu, sachez qu'il y a beaucoup à apprendre. Même un simple jeu est un challenge, et les jeux complexes ne sont pas nécessairement de bons jeux. Lorsque vous comprenez Python, vous pourrez utiliser Pygame pour créer un simple jeu en une ou deux semaines. Mais pour avoir un jeu complètement présentable, il vous faudra de nombreuses heures de travail.

Vue d'ensemble des modules de Pygame

Voici un récapitulatif des modules disponibles dans la bibliothèque de Pygame, avec un lien sur la documentation de référence de chacun de ces modules.

Cdrom (http://www.pygame.org/docs/ref/cdrom.html) Accéder et contrôler les lecteurs CD audio.	**Cursors (http://www.pygame.org/docs/ref/cursors.html)** Charger et compiler des images de curseur.	**Display (http://www.pygame.org/docs/ref/display.html)** Configurer la surface d'affichage.
Draw (http://www.pygame.org/docs/ref/draw.html) Dessiner des formes simples comme des lignes et des ellipses sur des surfaces.	**Event (http://www.pygame.org/docs/ref/event.html)** Gérer les évènements à partir de différents matériels d'entrée (clavier, souris, ...), ainsi que du fenêtrage.	**Font (http://www.pygame.org/docs/ref/font.html)** Charger et faire un rendu des polices TrueType.
Image (http://www.pygame.org/docs/ref/image.html) Charger, sauver et transférer des images sur des surfaces.	**Joystick (http://www.pygame.org/docs/ref/joystick.html)** Gérer les joysticks.	**Key (http://www.pygame.org/docs/ref/key.html)** Gérer le clavier.
Mixer (http://www.pygame.org/docs/ref/mixer.html) Charger et jouer des sons.	**Mouse (http://www.pygame.org/docs/ref/mouse.html)** Gérer la souris et son affichage.	**Movie (http://www.pygame.org/docs/ref/movie.html)**

		Lecture de vidéo à partir de film en MPEG.
Music (http://www.pygame.org/docs/ref/music.html) Jouer des pistes musicales en streaming.	**Overlay (http://www.pygame.org/docs/ref/overlay.html)** Accès à l'affichage vidéo avancé.	**Pygame (http://www.pygame.org/docs/ref/pygame.html)** Fonctions de haut niveau pour le contrôle de Pygame.
Rect (http://www.pygame.org/docs/ref/rect.html) Conteneur flexible pour un rectangle.	**Sndarray (http://www.pygame.org/docs/ref/sndarray.html)** Manipuler des échantillons de données audio.	**Sprite (http://www.pygame.org/docs/ref/sprite.html)** Objets de haut niveau pour la représentation des images de jeu.
Surface (http://www.pygame.org/docs/ref/surface.html) Objets pour des images et l'écran.	**Surfarray (http://www.pygame.org/docs/ref/surfarray.html)** Manipuler des données de pixel d'image.	**Time (http://www.pygame.org/docs/ref/time.html)** Manipuler les temporisateurs et le taux d'image.
Transform (http://www.pygame.org/docs/ref/transform.html) Redimensionner et déplacer des images.		

Importation et initialisation

Traduit de l'anglais, l'original par *Pete Shinners* :
http://www.pygame.org/docs/tut/ImportInit.html

Importer Pygame et l'initialiser est très facile. Elle est également assez souple pour nous laisser le contrôle sur ce qui se produit. Pygame est une collection de différents modules dans un simple paquet python. La plupart des modules sont écrits en C, d'autres sont écrits en Python. Certains modules sont en option, et ne sont pas toujours présents.

Ceci n'est qu'une courte introduction sur ce qui se passe lorsque vous importez Pygame. Pour des explications plus claires voyez les exemples sur Pygame.

Importation

Premièrement nous importons Pygame, c'est essentiel. Depuis la version 1.4, Pygame est fait pour être plus simple. La plupart des jeux importeront Pygame de cette façon :

```
import pygame
from pygame.locals import *
```

La première ligne est la seule obligatoire. C'est l'importation de tous les modules de Pygame existants. La seconde ligne est optionnelle et sert à rendre publiques certaines constantes et fonctions de Pygame.

Une chose à garder à l'esprit est que certains modules de Pygame sont optionnels. Par exemple le module `pygame.font` est un de ceux-ci. Lorsque le programme lit `import pygame`, Pygame vérifie si tous les modules sont disponibles. Si le module `font` est disponible, il est importé en tant que `pygame.font`. Si le module n'est pas disponible, `pygame.font` aura la valeur `None`. Ce qui fait qu'il est facile de tester la présence du module `font`.

Initialisation

Ensuite pour pouvoir utiliser Pygame, vous avez besoin de l'initialiser. La façon habituelle de le faire est d'écrire :

```
pygame.init()
```

Ceci initialise tous les modules de la bibliothèque Pygame pour nous. Tous les modules de Pygame n'ont pas besoin d'être initialisés, mais ceux qui en ont besoin le sont automatiquement. Il est également aussi facile d'initialiser les modules un par un. Par exemple pour initialiser le module `font` :

```
pygame.font.init()
```

Notons que s'il y a une erreur quand nous initialisons avec `pygame.init()`, elle échouera de façon silencieuse. Si nous initialisons les modules manuellement, toutes les erreurs soulèverons une exception. Tout module peut être initialisé avec la fonction `get_init()` qui retourne `true` si le module est initialisé correctement.

Il est possible d'appeler la fonction `init()` plusieurs fois par module sans danger.

Quit

Les modules ont généralement une fonction `quit()` pour quitter en libérant la mémoire. Il n'y a pas besoin d'appeler cette fonction explicitement vu que Python libère de lui même la mémoire de tous les modules initialisés quand on quitte un programme.

Déplacer une image

Traduit de l'anglais, original par *Pete Shinners* :
http://www.pygame.org/docs/tut/MoveIt.html
(nécessite une relecture approfondie, les tournures de phrase sont franchement bancales)

La plupart des gens qui commencent la programmation graphique ont des difficultés pour trouver comment faire bouger une image à l'écran. Sans comprendre tous les concepts, cela peut être très déroutant. Vous n'êtes pas la première personne à être bloquée ici, je ferais de mon mieux pour vous apprendre les choses étape par étape. Nous allons même essayer de terminer avec des méthodes pour garder vos animations efficaces.

Notez que nous n'enseignerons pas la programmation Python dans cet article, ceci n'est qu'une introduction aux fonctions basiques de Pygame.

De simples pixels sur l'écran

Pygame possède une Surface d'affichage. C'est typiquement l'image visible à l'écran, et cette image est constituée de pixels. La principale façon de modifier ces pixels est d'appeler la fonction `blit()` : elle copie les pixels d'une image sur une autre.

C'est la première chose à comprendre. En appelant la fonction `blit()` d'une image sur l'écran, vous changez simplement la couleur des pixels de l'écran. Les pixels ne sont pas ajoutés ou déplacés, c'est seulement la couleur de certains pixels qui est modifiée. Ces images que vous *blitez* sur l'écran sont des Surfaces dans Pygame, mais elles ne sont aucunement connectées à la Surface d'affichage. Quand elles sont *blitées* sur l'écran, elles sont copiées sur la Surface d'affichage, mais vous avez toujours accès à l'image originale.

Avec cette brève description, peut-être pouvez-vous déjà comprendre ce que nécessite l'animation d'une image. En réalité, nous ne déplaçons rien. Nous faisons simplement un *blit* de l'image dans une nouvelle position. Mais avant de dessiner l'image dans une nouvelle position, il faut effacer l'ancienne. Autrement, l'image serait visible à deux places sur l'écran. En effaçant rapidement l'image et en la redessinant à un nouvel endroit, nous réalisons *l'illusion* du mouvement.

À travers le reste du tutoriel, nous décomposerons ce processus en étapes simples. Nous verrons également comment animer plusieurs images à l'écran en même temps. Vous avez probablement déjà des questions, par exemple : *comment effacer l'image avant de la redessiner dans une nouvelle position ?* Peut-être êtes-vous déjà totalement perdu ? Nous espérons que le reste de ce tutoriel pourra éclaircir certaines choses.

Retournons sur nos pas

Peut-être que ce concept de pixels et d'images est encore un peu étranger à vos yeux ? Bonne nouvelle, durant les prochaines sections nous utiliserons du code pour faire tout ce que nous voulons, il n'utilisera pas de pixels. Nous allons créer une petite liste de 6 nombres en Python, et imaginer qu'elle représente des graphismes fantastiques que nous pourrions visualiser sur l'écran. Et il est surprenant de s'apercevoir à quel point cela correspond à ce que nous ferons plus tard avec des graphismes réels.

Alors commençons par créer notre liste et remplissons-la d'un beau paysage fait de 1 et de 2.

```
>>> screen = [1, 1, 2, 2, 2, 1]
>>> print screen
[1, 1, 2, 2, 2, 1]
```

Nous venons de créer l'arrière-plan. Mais ça ne sera pas franchement excitant tant que nous n'aurons pas dessiné un joueur à l'écran. Nous allons créer un puissant Héros qui ressemblera à un 8. Déposons-le au milieu de la carte et voyons de quoi il a l'air.

```
>>> screen[3] = 8
>>> print screen
[1, 1, 2, 8, 2, 1]
```

Vous n'êtes sûrement pas parvenus plus loin si vous avez tout juste commencé à faire de la programmation graphique avec pygame. Vous avez obtenu quelques trucs mignons sur l'écran, mais ils ne pouvaient se déplacer nulle part. Peut-être que maintenant que notre écran n'est qu'une simple liste de nombres, il est plus facile de voir comment les déplacer ?

Déplacement de notre Héros

Avant que nous puissions déplacer notre personnage, nous avons besoin de garder une trace de sa position. Dans la section précédente, quand nous l'avons dessiné, nous l'avons juste posé à une position arbitraire. Faisons-le plus rigoureusement cette fois-ci.

```
>>> playerpos = 3
>>> screen[playerpos] = 8
>>> print screen
[1, 1, 2, 8, 2, 1]
```

Maintenant il est assez facile de le déplacer vers une nouvelle position. Changeons simplement la valeur de `playerpos`, et dessinons-le une nouvelle fois à l'écran.

```
>>> playerpos = playerpos - 1
>>> screen[playerpos] = 8
>>> print screen
[1, 1, 8, 8, 2, 1]
```

Aïe! Maintenant nous pouvons voir 2 héros. Un dans l'ancienne position, et un dans la nouvelle. C'est exactement la raison pour laquelle nous avons besoin *d'effacer* le héros dans son ancienne position avant de le dessiner sur sa nouvelle position. Pour l'effacer, nous devons changer la valeur dans la liste pour qu'elle soit de nouveau comme avant la présence du héros. Pour ça, nous devons conserver une trace des valeurs de l'affichage avant que notre héros ne les remplace. Il y a plusieurs manières de le faire, mais la plus simple est de garder une copie séparée de l'arrière-plan. Ceci signifie que nous devons faire subir quelques modifications à notre jeu.

Création d'une carte

Ce que nous voulons faire c'est créer une liste séparée que nous appellerons arrière-plan. Nous créerons cet arrière-plan de façon à ce qu'il soit comme notre écran original rempli de 1 et de 2. Ensuite nous copierons chaque objet dans l'ordre de d'affichage : de l'arrière-plan vers l'écran. Après, nous pourrons redessiner notre héros sur l'écran.

```
>>> background = [1, 1, 2, 2, 2, 1]
>>> screen = [0]*6                        #Un nouvel écran vierge
>>> # Copie de l'arrière-plan sur l'écran
>>> for i in range(6):
...     screen[i] = background[i]
>>> print screen
[1, 1, 2, 2, 2, 1]
>>> # Positionnement du héros sur l'écran
>>> playerpos = 3
>>> screen[playerpos] = 8
>>> # Affichage du résultat
>>> print screen
[1, 1, 2, 8, 2, 1]
```

Cela peut sembler être un surplus de travail. Nous n'en sommes pas plus loin d'où nous étions la dernière fois, lorsque nous avons tenté de le déplacer. Mais cette fois nous avons plus d'information pour pouvoir le déplacer correctement.

Déplacement de notre Héros (2ème essai)

Cette fois ci, il sera plus simple de déplacer le héros. D'abord nous effacerons le héros de son ancienne position. Nous faisons cela en recopiant les bonnes valeurs de l'arrière-plan sur l'écran. Ensuite, nous dessinerons le personnage dans sa nouvelle position sur l'écran.

```
>>> print screen
[1, 1, 2, 8, 2, 1]
>>> screen[playerpos] = background[playerpos]
>>> playerpos = playerpos - 1
>>> screen[playerpos] = 8
>>> print screen
[1, 1, 8, 2, 2, 1]
```

Et voilà. Le héros s'est déplacé d'un pas vers la gauche. Nous pouvons utiliser le même code pour le bouger une nouvelle fois à gauche.

```
>>> screen[playerpos] = background[playerpos]
>>> playerpos = playerpos - 1
>>> screen[playerpos] = 8
>>> print screen
[1, 8, 2, 2, 2, 1]
```

Excellent! Ce n'est pas exactement ce que l'on pourrait appeler une animation fluide. Mais avec quelques arrangements, nous ferons ce travail directement avec des graphismes sur l'écran.

Définition de *blit*

Dans la prochaine partie, nous transformerons notre programme qui utilise des listes en un programme qui utilise de vrais graphismes. Pour l'affichage des graphismes, nous utiliserons le terme *blit* fréquemment. Si vous êtes débutant dans le graphisme, vous êtes probablement peu familier avec ce terme.

Blit
: À la base, un *blit* signifie copier le graphisme d'une image vers une autre. Une définition plus formelle serait copier un tableau de données source vers un tableau de données destination. Vous pouvez considérer qu'un *blit* n'est qu'une assignation de pixels. Comme définir des valeurs dans notre liste de nombres, *blit* assigne la couleur des pixels dans notre image.

D'autres bibliothèques graphiques utiliserons le termes *bitblt*, ou plus simplement *blt*, mais elles parlent de la même chose. C'est tout simplement copier la mémoire, d'un endroit à un autre. En fait, c'est plus complexe que ça puisqu'on a besoin de manipuler certaines choses comme l'espace colorimétrique, le découpage et le *scanline pitches*. Les *bliters* avancés peuvent utiliser certaines spécificités comme la transparence ou d'autres effets spéciaux.

De la liste à l'écran

Utiliser le code que nous avons vu dans les exemples plus haut, et le faire fonctionner avec Pygame est très simple :

1. Nous supposons que nous avons chargé de jolies images et que nous les avons nommées : `terrain1`, `terrain2` et `hero`.
2. Où nous avons assigné plus tôt des nombres à une liste, maintenant nous allons *bliter* des images à l'écran.
3. Un autre grand changement, au lieu d'employer des positions en tant que simple index (de 0 à 5), nous aurons besoin de coordonnées à deux dimensions. Nous supposerons également que chaque image de notre jeu aura une largeur de 10 pixels donc avec des positions multiples de 10. Cela revient à multiplier les indices par 10 pour obtenir les coordonnées.

```
>>> background = [terrain1, terrain1, terrain2, terrain2,
terrain2, terrain1]
>>> screen = create_graphics_screen()           #Un nouvel écran
```

```
vierge
>>> # Copie de l'arrière-plan sur l'écran
>>> for i in range(6):
...     screen.blit(background[i], (i*10, 0))
>>> # Positionnement du héros sur l'écran
>>> playerpos = 3
>>> screen.blit(playerimage, (playerpos*10, 0))
```

Ce code devrait vous sembler très familier, et peut-être même plus encore : le code ci-dessus devrait prendre un peu de sens. J'espère que mon illustration sur le paramétrage de valeurs simples dans une liste montre la similarité avec le paramétrage de pixels sur l'écran (avec blit). La seule partie qui soit un travail supplémentaire est celle qui convertit la position du joueur en coordonnée sur l'écran. Pour l'instant nous utilisons simplement `(playerpos*10, 0)`, mais nous pouvons certainement faire mieux que ça.

Maintenant déplaçons l'image du joueur dans un autre endroit. Ce code ne devrait pas vous surprendre.

```
>>> screen.blit(background[playerpos], (playerpos*10, 0))
>>> playerpos = playerpos - 1
>>> screen.blit(playerimage, (playerpos*10, 0))
```

Voila! Avec ce code, nous avons vu comment afficher un simple arrière-plan avec l'image du héros dessus. Ensuite nous avons correctement déplacé le héros d'un espace vers la gauche.

Et qu'allons nous faire maintenant ? Ce code est encore un peu maladroit. La première chose que nous voudrions faire serait de trouver une manière plus propre de représenter l'arrière-plan et la position du joueur. Et peut être de faire une vraie animation fluide.

Coordonnées écran

Pour positionner un objet sur l'écran, nous avons besoin de la fonction `blit()` où l'on met l'image. Dans Pygame nous passons toujours nos positions comme des coordonnées (X,Y). X est le nombre de pixels vers la droite et Y le nombre de pixels vers le bas. Le coin supérieur gauche d'une Surface correspond aux coordonnées `(0, 0)`. Un déplacement vers la droite donnerait `(10, 0)`, et ajouter un déplacement vers le bas nous donnerait `(10, 10)`. Quand nous blitons, l'argument passé en position représente le coin supérieur gauche de la source devant être placé sur la destination.

Pygame possède un conteneur intéressant, l'objet `Rect`. L'objet `Rect` représente une zone rectangulaire avec ses coordonnées. Il est défini par son coin supérieur gauche et sa dimension. L'objet `Rect` possède de nombreuses méthodes utiles qui peuvent vous aider à le déplacer et le positionner. Dans notre prochain exemple, nous représenterons les positions de nos objets avec des `Rect`.

Ensuite sachez que beaucoup de fonctions de Pygame utilisent les attributs des objets `Rect`. Toutes ces fonctions peuvent accepter un simple tuple de 4 éléments (position gauche, position dessus, largeur, hauteur). Vous n'êtes pas toujours obligé d'utiliser ces objets `Rect`, mais vous les trouverez utiles. Ainsi, la fonction `blit()` peut accepter un objet `Rect` en tant qu'argument de position, elle utilise le coin supérieur gauche du `Rect` comme position réelle.

Changer l'arrière-plan

Dans toutes nos sections précédentes, nous avons stocké l'arrière-plan comme étant une liste de différents types de sol. C'est une bonne manière de créer un jeu basé sur des cases, mais nous voulons faire un défilement (*scrolling* en anglais) fluide. Pour faire simple, nous commencerons par modifier l'arrière-plan en une simple image qui couvre entièrement l'écran. De cette façon, quand nous voudrons effacer nos objets (avant de les redessiner) nous aurons simplement besoin de bliter la section effacée de l'arrière-plan dans l'écran.

En passant un troisième argument optionnel à la fonction `blit()`, nous lui indiquerons de bliter uniquement une sous-section de l'image source. Vous la verrez en action quand nous effacerons l'image du joueur.

Notez également, que lorsque nous aurons fini de tout dessiner, nous invoquerons la fonction `pygame.display.update()` qui affichera tout ce que nous avons dessiné sur l'écran.

Mouvement fluide

Pour obtenir quelque chose qui apparaisse comme un mouvement fluide, nous déplacerons quelques pixels à la fois. Voici le code pour faire un objet qui se déplace de manière fluide à travers l'écran. Puisque basé sur ce que nous savons déjà, ceci devrait vous paraître simple.

```
>>> screen = create_screen()
>>> player = load_player_image()
>>> background = load_background_image()
>>> screen.blit(background, (0, 0))                    #dessiner
l'arrière-plan
>>> position = player.get_rect()
>>> screen.blit(player, position)                      #dessiner le
joueur
>>> pygame.display.update()                            #montrer le
tout
>>> for x in range(100):                               #animer 100
images
...     screen.blit(background, position, position)  #effacer
...     position = position.move(2, 0)               #déplacer le
joueur
...     screen.blit(player, position)                #dessiner un
nouveau joueur
...     pygame.display.update()                      #afficher le
tout
```

```
...     pygame.time.delay(100)                          #arrêter le
programme pour 1/10 secondes
```

Et voilà. Ceci correspond au code nécessaire pour animer de façon fluide un objet à travers l'écran. Nous pouvons aussi utiliser un joli personnage d'arrière-plan. Un autre avantage sur cette façon de procéder, est que l'image du joueur peut inclure de la transparence ou être découpée en section, elle sera toujours dessinée correctement sur l'arrière-plan.

Nous avons aussi fait appel à la fonction `pygame.time.delay()` à la fin de notre boucle. Ceci pour ralentir un peu notre programme, autrement il tournerait tellement vite et l'on aurait pas le temps de le voir.

Et ensuite ?

Si tout va bien, cet article a fait tout ce qu'il avait promis de faire. Mais à ce stade, le code n'est pas encore prêt pour réaliser le prochain jeu le plus vendu. Comment faire pour obtenir simplement de multiple déplacements d'objets ? Que sont réellement ces mystérieuses fonctions comme `load_player_image()` ? Nous avons également besoin d'une manière simple d'accéder aux entrées de l'utilisateur (clavier, souris ou autre), et boucler sur plus de 100 images. Nous prendrons l'exemple que nous avons ici, et le transformerons en une création orientée objet qui rendra notre maman très fière.

Les fonctions mystères

Des informations complètes sur ces types de fonctions peuvent être trouvées dans d'autres tutoriels et références. Le module `pygame.image` possède une fonction `load()` qui fera ce que nous voudrons. Les lignes pour charger des images devraient ressembler à ceci.

```
>>> player = pygame.image.load('player.bmp').convert()
>>> background = pygame.image.load('liquid.bmp').convert()
```

Nous pouvons voir la simplicité de l'exemple, la fonction de chargement demande seulement un nom de fichier et retourne une nouvelle surface avec l'image chargée. Après le chargement, nous faisons appel à la méthode de Surface : `convert()`. `convert()` nous retourne une nouvelle Surface contenant l'image, mais convertie dans le même espace colorimétrique que notre affichage. Maintenant que les images ont le même format d'affichage, le blit est très rapide. Si nous ne faisons pas la conversion, la fonction `blit()` est plus lente, c'est pourquoi il est préférable de faire la conversion de pixels d'un format à un autre au fur et à mesure.

Vous avez certainement dû remarquer que les deux fonctions, `load()` et `convert()`, retournent de nouvelles Surfaces. Ceci signifie que nous avons réellement créé deux Surfaces à chacune de ces lignes. Dans d'autres langages de programmation, on obtiendrait une fuite de mémoire (ce n'est clairement pas une bonne chose). Heureusement Python est suffisamment intelligent pour les gérer, et Pygame effacera proprement la Surface que nous n'utiliserons pas.

Cette autre fonction mystérieuse que nous avons vue dans l'exemple précédent était `create_screen()`. Dans Pygame, c'est simple de créer une nouvelle fenêtre pour les graphismes. Le code pour créer une surface de 640X480 pixels est le suivant. Sans passer aucun autre argument, Pygame choisit la meilleure profondeur de couleur et le meilleur espace colorimétrique pour nous.

```
>>> screen = pygame.display.set_mode((640, 480))
```

Manipulation des entrées utilisateur

Nous avons désespérément besoin de modifier la boucle principale pour prendre en compte une entrée utilisateur, comme par exemple, lorsque celui ci ferme la fenêtre. Nous devons ajouter la manipulation d'évènements à notre programme. Tous les programmes graphiques utilisent ce concept basé sur les évènements. Le programme reçoit des évènements de l'ordinateur lorsqu'une touche du clavier est enfoncée ou lorsque la souris s'est déplacée. Alors le programme répond aux différents évènements. Voici ce à quoi devrait ressembler le code. Au lieu de boucler sur 100 images, nous continuons à boucler jusqu'à ce que l'utilisateur nous demande d'arrêter.

```
>>> while 1:
...     for event in pygame.event.get():
...         if event.type in (QUIT, KEYDOWN):
...             sys.exit()
...     move_and_draw_all_game_objects()
```

Ce que fait ce code est, d'abord de boucler en continu, et ensuite de vérifier s'il y a un quelconque évènement provenant de l'utilisateur. Nous quittons le programme si l'utilisateur appuie sur un bouton de son clavier ou clique sur le bouton de fermeture de la fenêtre. Ensuite avoir vérifié tous les évènements, nous déplaçons et dessinons tous les objets du jeu. (Nous les effacerons également avant de les déplacer).

Déplacer de multiples images

Voici la partie où nous allons vraiment changer les choses. Disons que nous désirons déplacer 10 images différentes en même temps à l'écran. Une bonne manière de le faire est d'utiliser les classes Python. Nous allons créer une classe qui représente un objet du jeu. Cet objet aura une fonction pour se déplacer lui-même, nous pourrons alors en créer autant que nous le voulons. Les fonctions pour dessiner et déplacer cet objet nécessitent de travailler d'une manière où ils se déplacent seulement d'une image (ou d'un pas) à la fois. Voici le code de python pour créer notre classe.

```
>>> class GameObject:
...     def __init__(self, image, height, speed):
...         self.speed = speed
...         self.image = image
...         self.pos = image.get_rect().move(0, height)
...     def move(self):
...         self.pos = self.pos.move(0, self.speed)
...         if self.pos.right > 600:
...             self.pos.left = 0
```

Nous avons donc deux fonctions dans notre classe. La méthode `__init__()` construit notre objet. Elle le positionne et définit sa vitesse. La méthode `move()` bouge l'objet d'un pas. S'il va trop loin, elle déplace l'objet en arrière vers la gauche.

Positionner le tout

Maintenant avec notre nouvelle classe, nous pouvons assembler l'intégralité du jeu. Voici à quoi ressemblerait la fonction principale de notre programme.

```
>>> screen = pygame.display.set_mode((640, 480))
>>> player = pygame.image.load('player.bmp').convert()
>>> background = pygame.image.load('background.bmp').convert()
>>> screen.blit(background, (0, 0))
>>> objects = []
>>> for x in range(10):                    #Créer 10 objets
...     o = GameObject(player, x*40, x)
...     objects.append(o)
>>> while 1:
...     for event in pygame.event.get():
...         if event.type in (QUIT, KEYDOWN):
...             sys.exit()
...     for o in objects:
...         screen.blit(background, o.pos, o.pos)
...     for o in objects:
...         o.move()
...         screen.blit(o.image, o.pos)
...     pygame.display.update()
...     pygame.time.delay(100)
```

Ceci est le code dont nous avons besoin pour animer 10 objets à l'écran. Le seul point qui ait besoin d'explication, ce sont les deux boucles que nous utilisons pour effacer tous les objets et dessiner tous les objets. De façon à faire les choses proprement, nous avons besoin d'effacer tous les objets avant de redessiner chacun d'eux. Dans notre exemple nous n'avons pas de problème, mais quand les objets se recouvrent, l'utilisation de deux boucles comme celles-ci devient nécessaire. Autrement l'effacement de l'ancienne position d'un objet pourrait effacer la nouvelle position d'un objet affiché avant.

Les mots de la fin

Alors quelle sera la prochaine étape sur la route de votre apprentissage ? D'abord, jouer un peu avec cet exemple. La version complète de cet exemple est disponible dans le répertoire `examples` de Pygame, sous le nom `moveit.py`. Jetez un coup d'œil sur le code et jouez avec, lancez-le et apprenez-le.

Il y a plusieurs choses que vous pouvez faire avec, comme utiliser plus d'un type d'objet. Trouvez une façon pour supprimer proprement les objets quand vous ne désirez plus les afficher. Pour faire une mise à jour, utilisez la méthode `display.update()` pour passer une liste de zones d'écran qui ont changé.

Il existe beaucoup d'autres tutoriels et exemples pour Pygame qui peuvent vous aider en toutes circonstances. Alors maintenant, pour retenir en apprenant, retenez en lisant. : ^)

Enfin, vous êtes libre de vous inscrire sur la mailing-list de Pygame ou dans un salon de discussion et poser toutes les questions que vous voulez à ce sujet. Il y a toujours quelqu'un pour vous aider.

Pour finir, prenez du plaisir, c'est pour ça que les jeux sont faits !

Chimp - Ligne par ligne

Traduit de l'anglais, original par *Pete Shinners* :

http://www.pygame.org/docs/tut/chimp/ChimpLineByLine.html

Introduction

Dans les exemples de Pygame, il y a un petit exemple nommé *chimp*. Cet exemple simule un singe à frapper qui bouge dans un petit écran avec des promesses de récompenses. Cet exemple est en lui-même vraiment simple, et réduit la recherche d'erreurs dans le code. Cet exemple de programme démontre les possibilités de Pygame, comme la création de fenêtres graphiques, le chargement de fichiers d'images et de sons, de rendu de texte TTF (police TrueType), et la gestion des évènements de bases et des mouvements de la souris.

Ce programme ainsi que les images sont disponibles dans les sources de Pygame. Pour la version 1.3 de Pygame, cet exemple a été complètement réécrit pour ajouter quelques fonctions et corriger quelques erreurs. Ceci fait que la taille de l'exemple a doublé par rapport à l'original, mais nous donne plus de matière à analyser, aussi bon que soit le code, je ne peux que vous recommander de le réutiliser pour vos propres projets.

Ce tutoriel analyse le code bloc par bloc, expliquant comment le code fonctionne. Il sera également mentionné la façon dont le code pourrait être amélioré et quel contrôle d'erreur peut nous venir en aide.

Ceci est un excellent tutoriel pour les personnes qui étudient pour la première fois du code Pygame. Une fois Pygame complètement installé, vous pourrez trouver et exécuter vous-même la démo de *chimp* dans le répertoire des exemples.

Importer des modules

Voici le code qui importe tous les modules nécessaires dans notre programme. Il vérifie la disponibilité de certains des modules optionnels de Pygame.

```
import os, sys
import pygame
from pygame.locals import *

if not pygame.font: print 'Attention, polices désactivées'
if not pygame.mixer: print 'Attention, son désactivé'
```

D'abord, nous importons les modules standards de Python `os` et `sys`. Ceux-ci nous permettent de faire certaines choses comme créer des chemins de fichiers indépendants du système d'exploitation.

Dans la ligne suivante, nous importons l'ensemble des modules de Pygame. Quand Pygame est importé, tous les modules appartenant à Pygame sont importés. Certains modules sont optionnels, s'ils ne sont pas trouvés, leur valeur est définie à `None`.

Il existe un module Pygame spécial nommé `locals`. Ce module contient un sous-ensemble de Pygame. Les membres de ce module utilisent couramment des constantes et des fonctions qui ont prouvé leur utilité à être incorporé dans l'espace de nom global de votre programme. Ce module de locales inclut des fonctions comme `Rect()` pour un objet rectangle, et plusieurs constantes comme `QUIT, HWSURFACE` qui sont utilisées pour interagir avec le reste de Pygame. L'importation de ce module de locales dans l'espace de nom global est complètement optionnel. Si vous choisissez de ne pas l'importer, tous les membres des locales sont toujours disponibles dans le module `pygame`.

Enfin, nous avons décidé d'imprimer un joli message si les modules `font` ou `sound` ne sont pas disponibles dans Pygame.

Chargement des Ressources

Ici nous avons deux fonctions que nous pouvons utiliser pour charger des images et des sons. Nous examinerons chaque fonction individuellement dans cette section.

```
def load_image(name, colorkey=None):
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
    except pygame.error, message:
        print "Impossible de charger l'image :", name
        raise SystemExit, message
    image = image.convert()
    if colorkey is not None:
        if colorkey is -1:
            colorkey = image.get_at((0,0))
        image.set_colorkey(colorkey, RLEACCEL)
    return image, image.get_rect()
```

Cette fonction prend le nom de l'image à charger. Elle prend également un argument optionnel qui peut être utilisé pour définir une couleur clé à l'image. Une couleur clé est utilisée dans les graphismes pour représenter une couleur de l'image qui devra être transparente.

La première chose que cette fonction fait, est de créer un chemin de fichier complet vers le fichier. Dans cet exemple, toutes les ressources sont situées dans le sous-répertoire `data`. En utilisant la fonction `os.path.join()`, un chemin sera créé qui fonctionnera quelle que soit la plateforme sur laquelle est lancé le jeu.

Ensuite, nous chargeons l'image en utilisant la fonction `pygame.image.load()`. Nous enveloppons cette fonction dans un bloc de `try/except`, ainsi s'il y a un problème lors du chargement de l'image, nous pouvons quitter élégamment. Après que l'image soit chargée, nous faisons un appel important à la fonction `convert()`. Ceci crée une nouvelle copie de la Surface et la convertit dans un format et une profondeur de couleurs qui correspondent à l'affichage en cours. Ceci signifie que le blitage de l'image sur l'écran sera aussi rapide que possible.

Enfin, nous définissons la couleur clé de l'image. Si l'utilisateur fournit un argument pour la couleur clé, nous utiliserons cette valeur de couleur clé pour l'image. Ceci devrait être habituellement une valeur RGB, comme (255, 255, 255) pour le blanc. Vous pouvez également passer la valeur `-1` comme couleur clé. Dans ce cas, la fonction examinera la couleur en haut à gauche de l'image, et utilisera cette couleur comme couleur clé.

```
def load_sound(name):
    class NoneSound:
        def play(self): pass
    if not pygame.mixer:
        return NoneSound()
    fullname = os.path.join('data', name)
    try:
        sound = pygame.mixer.Sound(fullname)
    except pygame.error, message:
        print 'Impossible de charger le son :', wav
        raise SystemExit, message
    return sound
```

Vient ensuite la fonction de chargement de fichier son. La première chose que cette fonction fait est de contrôler si le module `pygame.mixer` a été importé correctement. Si non, elle retourne une petite instance de classe qui possède une méthode de lecture factice. Ceci agira comme un objet Son normal pour ce jeu qui tournera sans contrôle d'erreur supplémentaire.

Cette fonction est similaire à la fonction de chargement d'image, mais gère des problèmes différents. En premier lieu nous créons un chemin complet vers le fichier son, et chargeons ce fichier son à travers un bloc `try/except`, qui nous retourne alors l'objet Son chargé.

Classes d'objet du Jeu

Ici nous créons deux classes qui représentent les objets dans notre jeu. La plupart de la logique de jeu vient de ces deux classes. Nous les examinerons dans cette section.

```
class Fist(pygame.sprite.Sprite):
    """Déplacer un poing fermé sur l'écran qui suit la souris"""
    def __init__(self):
```

```
        pygame.sprite.Sprite.__init__(self)        #Appel du
constructeur de Sprite
        self.image, self.rect = load_image('fist.bmp', -1)
        self.punching = 0

    def update(self):
        "Déplace le poing sur la position de la souris"
        pos = pygame.mouse.get_pos()
        self.rect.midtop = pos
        if self.punching:
            self.rect.move_ip(5, 10)

    def punch(self, target):
        "Renvoie true si le poing entre en collision avec la
cible"
        if not self.punching:
            self.punching = 1
            hitbox = self.rect.inflate(-5, -5)
            return hitbox.colliderect(target.rect)

    def unpunch(self):
        "Appelé pour faire revenir le poing"
        self.punching = 0
```

Ici nous créons une classe pour représenter le poing du joueur. Elle est dérivée de la classe Sprite inclue dans le module `pygame.sprite`. La méthode `__init__()` est appelée lorsqu'une nouvelle instance de cette classe est créée. La première chose que nous faisons est de s'assurer d'appeler la méthode `__init__()` de notre classe de base. Ceci autorise la méthode `__init__()` de Sprite à préparer notre objet pour l'utiliser comme un sprite. Ce jeu utilise un des groupes de classes de dessin de sprite. Ces classes peuvent dessiner des sprites qui possèdent un attribut `image` et `rect`. En changeant tout simplement ces deux attributs, le moteur de rendu dessinera les images actuelles à leur position actuelle.

Tous les sprites possède une méthode `update()`. Cette méthode est généralement appelée une fois par image. C'est le lieu où vous pouvez mettre le code qui déplace et actualise les variables de chaque sprite. La méthode `update()` pour le poing déplace ce poing vers l'endroit où pointe la souris. Elle compense légèrement la position du poing si celui est en état de *frappe*.

Les deux fonctions suivantes `punch()` et `unpunch()` modifie l'état du poing. La méthode `punch()` retourne la valeur `true` si le poing entre en collision avec le sprite cible.

```
class Chimp(pygame.sprite.Sprite):
    """Déplace un singe à travers l'écran. Elle peut faire
tournoyer
    le singe quand il est frappé."""
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)        #Appel du
constructeur de Sprite
        self.image, self.rect = load_image('chimp.bmp', -1)
        screen = pygame.display.get_surface()
```

```
        self.area = screen.get_rect()
        self.rect.topleft = 10, 10
        self.move = 9
        self.dizzy = 0

    def update(self):
        "Déplace ou fait tournoyer, suivant l'état du singe"
        if self.dizzy:
            self._spin()
        else:
            self._walk()

    def _walk(self):
        "Déplacer le singe à travers l'écran, et le faire
pivoter à la fin"
        newpos = self.rect.move((self.move, 0))
        if not self.area.contains(newpos):
            if self.rect.left < self.area.left or \
                self.rect.right > self.area.right:
            self.move = -self.move
            newpos = self.rect.move((self.move, 0))
            self.image = pygame.transform.flip(self.image, 1, 0)
        self.rect = newpos

    def _spin(self):
        "Faire tournoyer l'image du singe"
        center = self.rect.center
        self.dizzy += 12
        if self.dizzy >= 360:
            self.dizzy = 0
            self.image = self.original
        else:
            rotate = pygame.transform.rotate
            self.image = rotate(self.original, self.dizzy)
        self.rect = self.image.get_rect(center=center)

    def punched(self):
        "Entraine le tournoiement du singe"
        if not self.dizzy:
            self.dizzy = 1
            self.original = self.image
```

La classe Chimp fait un peu plus de travail que celle du poing, mais rien de complexe. Cette classe déplacera le chimpanzé de gauche à droite sur l'écran. Quand le singe sera frappé, il tournoiera sur lui-même dans un superbe effet. Cette classe est dérivée de la classe de base Sprite, et est initialisée de la même façon que celle du poing. Pendant l'initialisation, la classe définit l'attribut `area` comme dimension de l'affichage.

La fonction `update()` du singe vérifie simplement l'état actuel, lequel est `true` quand le singe tournoie après un coup de poing. Elle appelle la méthode `_spin` ou `_walk`. Ces fonctions sont préfixées d'un underscore. C'est un idiome Python qui suggère que ces méthodes devraient uniquement être utilisées à l'intérieur de la classe `Chimp`. Nous pourrions aller plus loin en leur attribuant un double underscore, qui indiquera à Python de réellement essayer d'en faire des méthodes privées, mais nous n'avons pas besoin de ce type de protection. :)

La méthode `_walk` crée une nouvelle position pour le singe, en déplaçant le `rect` actuel d'un déplacement élémentaire. Si cette nouvelle position se situe à l'extérieur de la zone d'affichage de l'écran, elle inverse le déplacement élémentaire. Elle inverse également le sens de l'image en utilisant la fonction `pygame.transform.flip()`. C'est un effet rudimentaire qui permet au singe d'inverser le sens de son image suivant son déplacement.

La méthode `_spin()` est appelée quand le singe est *étourdi* (`dizzy`). L'attribut `dizzy` est utilisé pour enregistrer le nombre de rotation actuel. Quand le singe a entièrement tournoyé sur lui-même (360 degrés), il réinitialise l'image du singe à sa version droite originale. Avant d'appeler la fonction `transform.rotate()`, vous verrez que le code crée une référence locale à la fonction nommée `rotate()`. Il n'y a pas lieu de la faire pour cet exemple, nous l'avons uniquement fait ici pour conserver une longueur raisonnable à la ligne suivante. À noter qu'en appelant la fonction `rotate()`, nous faisons toujours tounoyer l'image originale du singe. Pendant la rotation, il y a une légère perte de qualité. Effectuer une rotation de façon répétitive sur la même image entraîne au fur et à mesure une dégradation de l'image. Quand une image tourne sur elle-même, la dimension de cette image sera modifiée. Ceci est dû au fait que les coins de l'image sortent de la dimension originale pendant la rotation, et augmente alors les dimensions de l'image. Nous nous assurons que le centre de la nouvelle image correspond au centre de l'ancienne image, de cette façon elle tournoie sans se déplacer.

La dernière méthode `punched()` indique que le sprite entre dans son état *étourdi* (`dizzy`). Ceci entrainera le tournoiement de l'image. Elle fera également une copie de l'actuelle appelée `original`.

Tout initialiser

Avant d'aller plus loin avec Pygame, nous devons nous assurer que tous ces modules sont initialisés. Dans ce cas nous ouvrirons une simple fenêtre graphique. Maintenant nous sommes dans la fonction principale du programme, laquelle exécute tout.

```
pygame.init()
screen = pygame.display.set_mode((468, 60))
pygame.display.set_caption('Monkey Fever')
pygame.mouse.set_visible(0)
```

La première ligne d'initialisation de Pygame nous épargne un peu de travail. Elle contrôle les modules Pygame importés et tente d'initialiser chacun d'entre eux. Il est possible de vérifier si des modules n'ont pas échoué pendant l'initialisation, mais nous ne nous tracasserons pas avec ça. Il est ainsi possible d'effectuer un contrôle plus fin et d'initialiser chaque module spécifique à la main. Ce type de contrôle n'est généralement pas indispensable, mais il est disponible si besoin.

Ensuite, nous définissons le mode d'affichage. À noter que le module `pygame.display` est utilisé pour contrôler tous les paramètres d'affichage. Dans cet exemple, nous demandons une simple fenêtre. Il existe un tutoriel complet sur le paramétrage du mode graphique, mais nous n'en avons pas réellement besoin,

Pygame effectue déjà un bon travail pour nous en obtenant quelque chose qui fonctionne. Pygame trouve la meilleure profondeur de couleur sans que nous lui en fournissions une.

Enfin nous définissons le titre de la fenêtre et désactivons le curseur de la souris de notre fenêtre. Très simple à faire, et maintenant nous avons une petite fenêtre noire prête à recevoir nos requêtes. En fait le curseur est par défaut visible, ainsi il n'y a pas réellement besoin de définir son état tant que nous ne voulons pas le cacher.

Créer l'arrière-plan

Notre programme affichera un message textuel en arrière-plan. Il serait bon pour nous de créer une simple surface pour représenter l'arrière-plan et l'utiliser à chaque fois. La première étape sera de créer cette surface.

```
background = pygame.Surface(screen.get_size())
background = background.convert()
background.fill((250, 250, 250))
```

Ceci crée pour nous une nouvelle surface qui est de la même taille que la fenêtre d'affichage. À noter l'appel supplémentaire `convert()` après la création de la surface. La méthode `convert()` sans argument s'assure que notre arrière-plan est du même format que la fenêtre d'affichage, ce qui nous donnera des résultats plus rapides.

Nous remplissons alors entièrement l'arrière-plan avec une couleur blanchâtre. La méthode `fill()` prend un triplet RGB en argument de couleur.

Appliquer le texte sur l'arrière-plan et le centrer

Maintenant que nous avons une surface d'arrière-plan, appliquons-lui un rendu de texte. Nous le ferons uniquement si nous voyons que le module `pygame.font` a été importé correctement. Si non, nous passons cette section.

```
if pygame.font:
    font = pygame.font.Font(None, 36)
    text = font.render("Pummel The Chimp, And Win $$$", 1, (10,
10, 10))
    textpos = text.get_rect(centerx=background.get_width()/2)
    background.blit(text, textpos)
```

Comme vous le voyez, il y a une paire d'étapes pour son obtention. D'abord nous créons un objet `font`, et en faisons un rendu sur la nouvelle surface. Nous trouvons alors le centre de cette nouvelle surface et la *blitons* sur l'arrière-plan.

La police est créée avec le constructeur `Font()` du module `font`. En fait, nous passons le nom du fichier de police truetype à cette fonction, mais nous pouvons aussi passer `None` et utiliser la police par défaut. Le constructeur `Font()` nécessite de connaître la taille de la police que nous désirons créer.

Nous faisons alors un rendu de cette police dans la nouvelle surface. La fonction `render()` crée une nouvelle surface d'une taille appropriée à notre texte. Dans cet exemple, nous dirons aussi à la fonction `render()` de créer un texte anti-aliasé (pour obtenir un effet lissé) et d'utiliser une couleur gris sombre.

Ensuite nous avons besoin de trouver le centre du texte sur l'affichage. Nous créons un objet `Rect` qui nous permet de l'assigner facilement au centre de l'écran.

Enfin, nous *blitons* le texte sur l'image d'arrière-plan.

Afficher l'arrière-plan une fois les paramètres définis

Nous avons encore une fenêtre noire sur l'écran. Affichons notre arrière-plan pendant que nous attendons de charger les autres ressources.

```
screen.blit(background, (0, 0))
pygame.display.flip()
```

Ceci *blitera* notre arrière-plan complet sur la fenêtre d'affichage. Le *blit* est lui-même trivial, mais qu'en est-il de la routine `flip()` ?

Dans Pygame, les changements de la surface d'affichage ne sont pas immédiatement visibles. Normalement, un affichage doit être mis à jour dans les zones qui ont changé pour les rendre visibles à l'utilisateur. Avec l'affichage par double-tampon (double buffer), l'affichage doit être interverti pour rendre les changements visibles. Dans cet exemple, la fonction `flip()` fonctionne parfaitement parce qu'elle manipule la zone entière de la fenêtre et gère les surfaces en simple et double tampon.

Préparer les objets du jeu

Ici nous créons tous les objets dont le jeu aura besoin.

```
whiff_sound = load_sound('whiff.wav')
punch_sound = load_sound('punch.wav')
chimp = Chimp()
fist = Fist()
allsprites = pygame.sprite.RenderPlain((fist, chimp))
clock = pygame.time.Clock()
```

D'abord nous chargeons deux effets sonores en utilisant la fonction `load_sound()`. Ensuite nous créons une instance pour chacune de nos classes de sprite. Et enfin, nous créons un groupe de sprites qui contiendra tous nos sprites.

Nous utilisons en fait un groupe spécial de sprites nommé `RenderPlain`. Ce sprite peut dessiner tous les sprites qu'il contient à l'écran. Il est appelé `RenderPlain` parce qu'il y a en fait plusieurs groupes de rendu avancés. Mais pour notre jeu, nous avons simplement besoin de les dessiner. Nous créons le groupe nommé `allsprites` en passant une liste avec tous les sprites qui appartiennent au groupe. Nous pourrons plus tard ajouter ou supprimer des sprites de ce groupe, mais dans ce jeu, nous n'en aurons pas besoin.

L'objet `clock` que nous créons, sera utilisé pour contrôler le taux d'images par seconde de notre jeu. Nous l'utiliserons dans la boucle principale de notre jeu pour s'assurer qu'il ne fonctionne pas trop vite.

La boucle principale

Rien de spécial ici, simplement une boucle infinie.

```
while 1:
    clock.tick(60)
```

Tous les jeux exécutent ce type de boucle. L'ordre des choses est de vérifier l'état de l'ordinateur et des entrées utilisateur, déplacer et actualiser l'état de tous les objets, et ensuite de les dessiner sur l'écran. Vous verrez que cet exemple n'est pas différent.

Nous faisons ainsi appel à notre objet `clock` qui s'assurera que notre jeu ne dépasse pas les 60 images par seconde.

Gérer tous les évènements d'entrée

C'est un exemple extrêmement simple sur le fonctionnement de la pile d'évènement.

```
for event in pygame.event.get():
    if event.type == QUIT:
        return
    elif event.type == KEYDOWN and event.key == K_ESCAPE:
        return
    elif event.type == MOUSEBUTTONDOWN:
        if fist.punch(chimp):
            punch_sound.play() #frappé
            chimp.punched()
        else:
            whiff_sound.play() #raté
    elif event.type == MOUSEBUTTONUP:
        fist.unpunch()
```

D'abord, nous prenons tous les évènements disponibles de Pygame et faisons une boucle pour chacun d'eux. Les deux premiers testent si l'utilisateur a quitté notre jeu ou a appuyé sur la touche `Echap`. Dans ces cas, nous retournons simplement dans la fonction principale et le programme se termine proprement.

Ensuite, nous vérifions si le bouton de la souris a été enfoncé ou relaché. Si le bouton est enfoncé, nous demandons à l'objet poing si il est entré en collision avec le chimpanzé. Nous jouons l'effet sonore approprié, et si le singe est frappé, nous lui demandons de tournoyer (en appelant sa méthode `punched()`.

Actualiser les Sprites

```
allsprites.update()
```

Les groupes de sprites possèdent une méthode `update()`, qui appelle la méthode `update()` de tous les sprites qu'ils contiennent. Chacun des objets se déplacera, en fonction de l'état dans lequel ils sont. C'est ici que le chimpanzé se déplacera d'un pas d'un côté à l'autre, ou tournoiera un peu plus loin s'il a récemment été frappé.

Dessiner la scène entière

Maintenant que tous les objets sont à la bonne place, il est temps de les dessiner.

```
screen.blit(background, (0, 0))
allsprites.draw(screen)
pygame.display.flip()
```

Le premier appel à `blit()` dessinera l'arrière-plan sur la totalité de la fenêtre. Ceci effacera tout ce que nous avons vu de la scène précédente (peu efficace, mais suffisant pour ce jeu). Ensuite nous appelons la méthode `draw()` du conteneur de sprites. Puisque ce conteneur de sprites est réellement une instance de groupe de sprites *DrawPlain*, il sait comment dessiner nos sprites. Enfin grâce à la méthode `flip()`, nous affichons le contenu du tampon de Pygame à l'écran. Tout ce que nous avons dessiné apparaît en une seule fois.

Game Over

L'utilisateur a quitté, c'est l'heure du nettoyage.

Le nettoyage d'un jeu dans Pygame est extrêmement simple. En fait puisque que toutes les variables sont automatiquement détruites, nous n'avons pas réellement besoin de faire quoi que ce soit.

Introduction au module Sprite

Traduit de l'anglais, l'original par *Pete Shinners* :
http://www.pygame.org/docs/tut/SpriteIntro.html

La version 1.3 de Pygame inclut un nouveau module, `pygame.sprite`. Ce module est écrit en python et inclut quelques classes à haut niveau d'abstraction pour gérer vos objets de jeu. En utilisant ce module à son plein potentiel, vous pouvez facilement gérer et dessiner vos objets de jeu. Les classes du module sprite sont très optimisées, il est donc probable que votre jeu fonctionne plus rapidement avec ce module que sans.

Le module sprite est censé être très générique. Il est possible de l'utiliser pour pratiquement n'importe quel type de gameplay. Cette grande flexibilité a toutefois un léger défaut, elle nécessite un peu de réflexion/compréhension pour pouvoir l'utiliser correctement. La documentation de référence (http://www.pygame.org/docs/ref/sprite.html) du module Sprite peut vous venir en aide, mais vous aurez probablement besoin d'un peu d'explications supplémentaires pour utiliser `pygame.sprite` dans votre jeu.

Plusieurs des exemples d'utilisation de Pygame (comme *chimp* et *aliens*) ont été mis à jour pour utiliser le module sprite. Vous voudrez certainement examiner ceux là d'abord pour observer ce qu'est approximativement le module sprite. Le module chimp possède de toutes façon son propre tutoriel en ligne par ligne, qui pourra en plus vous aider à comprendre la programmation avec Python et Pygame.

Tenez compte du fait que cette introduction suppose que vous ayez un peu d'expérience en programmation Python, et que vous êtes, d'une quelconque façon, familier avec les différentes étapes de création d'un simple jeu. Dans ce tutoriel, le mot *référence* est occasionnellement utilisé. Il représente une variable Python. Les variables avec Python sont des références, vous pouvez donc avoir plusieurs variables qui pointent vers le même objet.

Leçon d'histoire

Le terme *sprite* est une survivance des vieux ordinateurs et consoles de jeu. Ces vieilles boîtes étaient incapables de dessiner et d'effacer des graphismes suffisamment rapidement pour faire fonctionner des jeux. Ces machines possédaient un matériel spécial pour manipuler les objets du jeu qui avaient besoin d'être animés très rapidement. Ces objets étaient appelés *sprites* et avaient des limitations qui leur étaient propres, mais pouvaient être dessinés et mis à jour très rapidement. Ils étaient habituellement contenus dans des tampons spéciaux du circuit vidéo. De nos jours, les ordinateurs sont généralement assez rapides pour manipuler les sprites comme des objets standards sans nécessiter de matériel dédié. Le terme sprite est toujours utilisé pour parler des objets animés dans les jeux en deux dimensions.

Les classes

Le module sprite utilise deux classes principales. La première est *Sprite*, qui est censée être utilisée comme classe de base pour tous les objets du jeu. Cette classe ne fait pas grand chose en elle-même, elle inclut seulement plusieurs fonctions pour aider à la gestion des objets du jeu. L'autre classe est *Group*. La classe *Group* constitue un conteneur pour différents objets sprite. Il y a en réalité différents types de classes *Group*. Certains types de classe *Group* peuvent, par exemple, dessiner tous les éléments qu'elles contiennent.

C'est vraiment tout ce qu'il y a. Nous commencerons avec une description de ce que fait chaque type des classes, et nous discuterons sur la manière de les utiliser correctement.

La classe Sprite

Comme mentionné auparavant, la classe Sprite est conçue pour être une classe de base pour tous les objets de votre jeu. Vous ne pouvez pas vraiment l'utiliser en tant que telle, car elle est constituée seulement de plusieurs méthodes pour faciliter le travail avec les différentes classes de Group. Le sprite garde une trace du groupe auquel il appartient. Le constructeur de classe (la méthode `__init__()`) prend un argument de type *Group* (ou une liste d'arguments de type *Group*) que l'instance de Sprite devrait contenir. Vous pouvez également changer les membres d'un objet de type Group avec les méthodes `add()` et `remove()`. Il y a également une méthode `groups()`, qui retourne une liste des groupes actuels contenant le sprite

Lorsque vous utilisez les classes Sprite, il est préférable de les considérer comme *valides* ou *vivantes* lorsqu'elles sont contenues dans un ou plusieurs groupes. Lorsque vous supprimez l'instance de tous les groupes, Pygame va supprimer l'objet (sauf si vous possédez votre propre référence de l'objet quelque part). La méthode `kill()` supprime le sprite de tous les groupes qui le contiennent. Elle supprime proprement l'objet sprite. Si, par hasard, vous avez déjà réuni le code source de deux jeux, vous savez certainement que supprimer proprement un objet du jeu peut être ardu. Un sprite possède également une méthode `alive()`, qui retourne `true` s'il est encore membre d'au moins un groupe.

La classe Group

La classe Group est un simple conteneur. Comme sprite, elle possède une méthode `add()` et une méthode `remove()` qui peuvent modifier un groupe tant que des sprites y sont contenus. Vous pouvez également passer un sprite ou une liste de sprites au constructeur (méthode `__init__()`) pour créer une instance de Group qui contiendra des sprites initiaux.

Le groupe possède d'autres méthodes comme `empty()` pour supprimer tous les sprites du groupe et `copy()` qui va renvoyer une copie du groupe et de tous ses membres à l'identique. La méthode `has()` fait une vérification rapide sur le fait qu'un groupe contienne tel sprite ou telle liste de sprites.

L'autre fonction que vous utiliserez fréquemment est la méthode `sprites()`. Elle retourne un objet sur lequel il est possible de boucler pour avoir un accès à chacun des sprites que le groupe contient. Actuellement, c'est seulement une liste de sprites, mais dans les prochaines versions de Python, elle utilisera les itérateurs pour de meilleures performances. (NdT : n'utilise-t-elle pas déjà les itérateurs ?)

Tel un raccourci, la classe Group contient également une méthode `update()`, qui va appeler la méthode `update()` sur chacun des sprites du groupe, en passant les mêmes arguments à chacune d'elles. Habituellement dans un jeu, vous avez besoin d'une fonction qui met à jour l'état des objets. Il est très simple d'appeler vos propres méthodes en utilisant la méthode `Group.sprites()`, mais c'est un

raccourci suffisamment utilisé pour être inclus dans la classe Group. Remarquez également que la classe de base Sprite a une méthode `update()` vide qui prend n'importe quelle sorte d'argument et ne fait strictement rien.

Enfin, la classe Group a deux autres méthodes qui vous permettent d'utiliser la méthode interne `len()`, récupérant le nombre de sprites contenus. Et l'opérateur `truth`, qui vous permet d'écrire `if mygroup:` afin de vérifier si le groupe a des sprites ou non.

Utilisation couplée des deux classes

Arrivé là, les deux classes semblent vraiment simples. Elles ne font pas beaucoup plus que ce que vous pourriez faire avec une simple liste et votre propre classe d'objets du jeu. Mais il y a de gros avantages à utiliser les modules *Sprite* et *Group* ensemble. Un seul sprite peut être contenu dans autant de groupe que vous voulez. Rappelez-vous que dès qu'un sprite n'est plus contenu dans aucun groupe, il sera automatiquement supprimé (sauf si la référence de cet objet existe ailleurs que dans un groupe).

La première chose importante est que l'on a un moyen rapide et simple de séparer les sprites en catégories. Par exemple, supposons que l'on ait un jeu de type *pac-man*. Nous pourrions faire des groupes séparés pour les différents types d'objet dans le jeu, par exemple, un pour les fantômes, un pour pac-man et un pour les pilules. Quand pac-man mange une pilule de puissance, nous pouvons changer l'état de tous les fantômes en agissant sur le groupe des fantômes. C'est plus rapide et plus simple que de boucler à travers une liste de tous les objets du jeu et de vérifier si chacun d'entre eux est un fantôme, puis alors de le modifier.

Ajouter et déplacer des groupes et des sprites est une opération très rapide, plus rapide que d'utiliser des listes pour tout stocker. Ainsi, vous pouvez très efficacement changer les membres d'un groupe. Les groupes peuvent être utilisés pour fonctionner comme de simples attributs pour chaque objet du jeu. Au lieu de suivre un attribut comme `close_to_player` pour un groupe d'objets ennemis, vous pourriez les ajouter à un groupe séparé. Lorsque vous aurez besoin d'accéder à tous les ennemis qui sont proches du joueur, vous en aurez déjà la liste, au lieu de parcourir une liste de tous les ennemis, en vérifiant qu'elles ont l'attribut `close_to_player`. Plus tard dans votre jeu, vous pourrez ajouter plusieurs joueurs, et au lieu d'ajouter des attributs supplémentaires `close_to_player2`, `close_to_player3`, vous pourrez facilement ajouter différents groupes correspondant à chacun de ces joueurs.

Un autre bénéfice important apporté par l'utilisation des classes *Sprite* et *Group* est que les groupes permettent la suppression facile des objets du jeu. Dans un jeu où beaucoup d'objets font référence à d'autres objets, parfois, supprimer un objet peut être très difficile, parce que ça nécessite que sa référence ne soit plus contenue nul part. Supposons que nous ayons un objet qui en *poursuive* un autre. Le *poursuivant* pourra avoir alors défini un simple groupe qui fait référence à l'objet (ou aux objets) qu'il est en train de poursuivre. Si l'objet *poursuivi* est détruit, nous n'avons pas besoin de nous soucier de dire au *poursuivant* d'arrêter de le *poursuivre*. Le *poursuivant* pourra voir de lui même que son groupe de *poursuivis* est maintenant vide, et pourra chercher une nouvelle cible.

Encore une fois, la chose dont il faut se rappeler est que ajouter ou supprimer des sprites d'un groupe est une opération très peu consommatrice en temps de calcul. Vous pouvez gagner en efficacité en constituant directement plusieurs groupes pour contenir et organiser les objets du jeu. Ils peuvent rester inutilisés et

vides pour de grandes parties du jeu, vous n'avez pas besoin de vous en occuper, il n'y aura aucune contrepartie si vous créez directement tous les groupes dont vous pensez avoir besoin, sans les utiliser au premier abord.

Les différents types de groupe

Les exemples ci-dessus et les raisons d'utiliser Sprite et les groupes sont seulement la partie émergée de l'iceberg. Un autre avantage est que le module sprite possède différents types de groupes. Ces différents types héritent tous du type ancêtre Group, mais ils possèdent également des fonctionnalités supplémentaires (où touchent à des fonctionnalités différentes). Voici une liste des classes de type Group présentes dans le module sprite.

Group
: C'est le groupe de type standard, *sans supplément*, qui est explicité ci-dessus. La plupart des autres groupes sont dérivés de celui-ci, mais pas tous.

GroupSingle
: Celui-ci fonctionne exactement comme la classe *Group* standard, mais il contient seulement le sprite le plus récemment ajouté. Ainsi, lorsque vous ajoutez un sprite à ce groupe, il *oublie* tout du précédent sprite qui était stocké. De cette manière, un groupe de ce type contient toujours seulement un ou zéro sprites.

RenderPlain
: C'est un type dérivé du type Group. Il possède une méthode `draw()` qui dessine tous les sprites qu'il contient à l'écran (ou sur une surface quelconque). Pour ce faire, il nécessite que tous les sprites qu'il contient possèdent les attributs `image` et `rect`. Il les utilise pour savoir quoi bliter et où les bliter.

RenderClear
: Ce type est dérivé du type RenderPlain, et ajoute la méthode `clear()`. Cette méthode va effacer la position précédente de tous les sprites dessinés, en utilisant une image de fond pour les remplacer. Elle est suffisamment élégante pour gérer les sprites supprimés et les effacer proprement de l'écran lors de l'appel de la méthode `clear()`.

RenderUpdates
: C'est la cadillac des groupes de rendu. Il hérite de *RenderClear*, mais change la méthode `draw()` pour retourner une liste de `Rects` de Pygame, qui représente toutes les zones de l'écran qui ont été modifiées.

Voici la liste des différents types de groupe disponibles. Nous discuterons plus longuement de ces groupes de rendu dans la prochaine section. Il n'y a rien qui doive vous empêcher de créer votre propre classe héritée de *Group* de cette façon. Ces groupes sont seulement du code python, donc vous pouvez hériter d'un des ces types explicités précédemment et ajouter ou modifier n'importe lequel des attributs ou des méthodes. À l'avenir, j'espère que nous pourrons ajouter quelques types de groupe supplémentaires dans cette liste. Par exemple un GroupMulti qui est comme GroupSingle, mais pourra contenir un nombre fixe de sprites (dans une sorte de buffer circulaire ?). Également un super-render group qui pourra effacer la position d'anciens sprites sans avoir besoin d'une image de fond pour le faire (en mémorisant une copie de l'écran avant le blit). Qui sait, mais dans l'avenir de nouvelles classes utiles seront susceptibles d'être ajoutées à cette liste.

Les groupes de rendu

Précédemment, nous avons vu qu'il existe trois groupes de rendu différents. Nous pouvons probablement débuter avec le *RenderUpdates*, mais il intègre des manipulations supplémentaires qui ne sont pas réellement nécessaires pour un jeu en scrolling, par exemple. Nous avons donc ici plusieurs outils : à chaque type de travail correspond son outil.

Pour un jeu à scrolling, où l'arrière-plan change complètement à chaque image, nous n'avons manifestement pas besoin de nous soucier de la mise à jour des rectangles dans l'appel `display.update()`. Vous devriez en définitive, partir avec le groupe *RenderPlain* pour gérer votre rendu.

Pour les jeux où l'arrière-plan sera plus statique, vous ne voudriez pas que Pygame régénère entièrement l'écran (surtout qu'il n'y en a pas besoin). Ce type de jeu implique l'effacement de l'ancienne position de chaque objet, et son dessin à un nouvel endroit à chaque image. De cette façon, nous ne changeons que ce qui est nécessaire. La plupart du temps, vous utiliserez la classe *RenderUpdates* pour ces types de jeu, puisque vous voudrez passer cette liste de changements à la fonction `display.update()`.

La classe *RenderUpdates* effectue aussi un joli travail pour minimiser les zones qui se chevauchent dans la liste des rectangles actualisés. Si la position précédente et la position actuelle d'un objet se chevauchent, la classe *RenderUpdates* les fusionnera en un rectangle unique. Combinez ceci avec le fait qu'elle gère proprement les objets supprimés, et vous obtiendrez une classe de Groupe très puissante. Si vous avez écrit un jeu qui manipule les rectangles modifiés pour les objets dans un jeu, vous savez que c'est la cause d'une importante quantité de code *sale* dans votre jeu. Spécialement lorsque vous vous lancez dans les objets qui peuvent être effacés à chaque instant. Tout ce travail se réduit à une méthode `clear()` et `draw()` avec cette classe monstre. De plus avec le contrôle du chevauchement, c'est d'autant plus rapide que si vous le faisiez vous-même.

Sachez que rien ne vous empêche de mêler et d'associer ces groupes de rendu dans votre jeu. Vous pouvez finalement utiliser plusieurs groupes de rendu si vous voulez associer vos sprites en couches. Ainsi, si l'écran est séparé en plusieurs sections, peut-être que chaque section de l'écran devra utiliser un groupe de rendu approprié ?

Détection de collision

Le module sprite inclut deux fonctions de détection de collision très génériques. Pour des jeux trop complexes, elles ne seront pas suffisantes, mais vous pouvez facilement vous inspirer du code source et le modifier comme voulu. Voici un résumé de ces fonctions et de ce qu'elles font.

spritecollide(sprite, group, dokill) -> liste

> Celle-ci contrôle les collisions entre un sprite unique et les sprites d'un groupe. Elle requiert un attribut `rect` pour tous les sprites utilisés. Elle retourne la liste de tous les sprites qui chevauchent le premier sprite. L'argument `dokill` est un booléen. S'il est vrai, la fonction appelle la méthode `kill()` sur tous les sprites. Ceci signifie que la dernière référence de chaque sprite est probablement dans la liste retournée. Une fois la liste disparue, les sprites le seront aussi. Un exemple rapide sur son utilisation dans une boucle.

```
>>> for bomb in sprite.spritecollide(player, bombs, 1):
...     boom_sound.play()
...     Explosion(bomb, 0)
```

La fonction recherche tous les sprites du group `bomb` qui entrent en collision avec le joueur. À cause de l'argument `dokill`, elle supprime toutes les bombes écrasées. Pour chaque bombe entrée en collision avec le joueur, elle joue l'effet sonore `boom_sound` et crée une nouvelle explosion à l'endroit de la bombe. À noter ici que la classe Explosion sait ajouter chaque instance à la classe appropriée, ainsi nous n'avons pas besoin de l'enregistrer dans une variable, cette dernière ligne peut sembler légèrement *amusante* pour des programmeurs python.

groupcollide(group1, group2, dokill1, dokill2) -> dictionnaire

Celle-ci est similaire à la fonction `spritecollide()`, mais est un peu plus complexe. Elle vérifie les collisions de tous les sprites d'un groupe avec les sprites d'un autre. Il y a également un argument *dokill* pour les sprites de chaque liste. Quand `dokill1` est vrai, les sprites en collision dans le groupe 1 subiront la méthode `kill()`. Si `dokill2` est vrai, nous obtiendrons les mêmes résultats pour le groupe 2. Le dictionnaire qu'elle retourne fonctionne comme ceci : chaque clé du dictionnaire est un sprite du groupe 1 pour lequel il y a collision. La valeur de cette clé est la liste des sprites du groupe 2 qui sont en collision avec lui. Peut-être qu'un rapide exemple sera plus explicite.

```
>>> for alien in sprite.groupcollide(aliens, shots, 1, 1).keys()
...     boom_sound.play()
...     Explosion(alien, 0)
...     kills += 1
```

Ce code vérifie les collisions entre les balles du joueur et tous les aliens qu'elles ont croisés. Dans ce cas nous bouclons simplement par les clés du dictionnaire, mais nous pouvons aussi boucler par les valeurs ou les items si nous voulons faire quelque chose avec les balles entrées en collision avec les aliens. Si nous bouclons par les valeurs, nous voudrions boucler les listes qui contiennent les sprites. Le même sprite peut toutefois apparaître plus d'une fois dans ces différentes boucles, puisque la même balle peut être entrée en collision avec de multiples aliens.

Ce sont les fonctions de collision fournies avec Pygame. Il devrait être simple de créer vos propres fonctions qui pourraient utiliser quelquechose de différent que les attributs `rect`. Ou peut-être essayer d'affiner un peu plus votre code en affectant directement les objets en collision, au lieu de construire une liste de collision. Le code des fonctions de collision de sprite est très optimisé, mais vous pouvez légèrement l'accélérer en supprimant certaines fonctionnalités qui ne vous sont pas nécessaires.

Problèmes connus

Actuellement, il existe un problème principal soulevé par de nouveaux utilisateurs. Quand vous dérivez votre nouvelle classe de sprite, vous devez appeler la méthode `Sprite.__init__()` à partir de votre propre constructeur de classe `__init__()`. Si vous oubliez d'appeler la méthode `Sprite.__init__()`, vous obtiendrez une erreur assez énigmatique, du style : `AttributeError: 'mysprite' instance has no attribute '_Sprite__g'`.

Dériver vos propres classes (Expert)

Concernant la rapidité, les classes de groupes courantes essaient de faire exactement ce dont elles ont besoin, et ne gère pas énormément de situations générales. Si vous décidez que vous avez besoin de fonctionnalités spéciales, vous devriez créer votre propre classe de groupe.

Les classes *Sprite* et *Group* sont conçues pour être dérivées, n'hésitez pas à créer vos propres classes *Group* pour effectuer des actions spécialisées. La meilleure manière de commencer est probablement le code source du module de sprite. L'examen des groupes *Sprite* actuels devrait constituer un bon exemple sur la façon de créer les vôtres.

Par exemple, voici le code source pour un groupe de rendu qui appelle une méthode `render()` pour chaque sprite, au lieu de simplement bliter une variable *image*. Puisque nous voulons gérer uniquement les zones actualisées, nous démarrerons avec une copie du groupe original *RenderUpdate*, en voici le code :

```
class RenderUpdatesDraw(RenderClear):
    """Appel de sprite.draw(screen) pour faire un rendu des
sprites"""
    def draw(self, surface):
        dirty = self.lostsprites
        self.lostsprites = []
        for s, r in self.spritedict.items():
            newrect = s.draw(screen)          #Voici la
principale modification
            if r is 0:
                dirty.append(newrect)
            else:
                dirty.append(newrect.union(r))
            self.spritedict[s] = newrect
        return dirty
```

Following is more information on how you could create your own Sprite and Group objects from scratch.

Les objets Sprite requièrent uniquement deux méthodes : `add_internal()` et `remove_internal()`. Elles sont appelées par les classes *Group* quand elles s'enlèvent un sprite d'elles-mêmes. Les fonctions `add_internal()` et `remove_internal()` possède un unique argument qui est un Groupe. Votre Sprite aura besoin d'une technique pour conserver une trace des Groupes auxquels il appartient. Vous voudriez peut-être essayer d'assortir les autres méthodes et arguments avec la vraie classe Sprite, mais si vous n'avez pas l'utilité de ces méthodes, vous êtes certains de ne pas en avoir besoin.

C'est presque les mêmes exigences pour la création de votre propre groupe. En fait, si vous examinez le code source, vous verrez que le *GroupSingle* n'est pas dérivé de la classe *Group*, il implémente uniquement les mêmes méthodes, vous ne pouvez donc pas faire réellement la différence. Une fois encore, vous aurez besoin des méthodes `add_internal()` et `remove_internal()` que les sprites appellent lorsqu'ils veulent s'ajouter ou se retirer eux-mêmes du groupe. Les fonctions `add_internal()` et `remove_internal()` possède un unique argument qui est un sprite. La seule autre nécessité pour les classes groupes est d'avoir un attribut factice nommé `_spritegroup`. Sa valeur n'est pas importante, tant que cet attribut est présent. Les classes Sprite peuvent chercher cet attribut pour déterminer la différence

entre un *Group* et un conteneur Python ordinaire. C'est important, car plusieurs méthodes de sprite peuvent prendre un argument d'un groupe unique, ou une séquence de groupes. Puisque les deux sont similaires, c'est la manière la plus flexible de *voir* la différence.

Vous devriez lire le code source du module sprite. Là où le code est un peu *customisé*, il y a suffisamment de commentaires pour vous aider à suivre. Il existe également une section *todo* dans le code source, si vous voulez contribuer.

Introduction au module Surfarray

Traduit de l'anglais, original par *Pete Shinners* :

http://www.pygame.org/docs/tut/surfarray/SurfarrayIntro.html

Introduction

Ce tutoriel a pour objectif d'introduire les utilisateurs à Numeric et au module Surfarray de Pygame. Pour les débutants, le code utilisé par Surfarray peut être légèrement intimidant. Mais ici, il y a seulement quelques concepts à comprendre et vous serez opérationnel. En utilisant le module Surfarray, il devient possible de réaliser des opérations au niveau du pixel en utilisant du code python pur. Les compétences requises pour faire cela en C sont d'un niveau beaucoup plus difficilement accessible.

Vous pouvez avoir envie d'aller directement voir à la section Exemples pour vous faire une idée sur ce qu'il est possible de faire avec ce module, ensuite nous commencerons par le début pour vous montrer la manière d'y arriver.

Maintenant, je ne vais pas essayer de vous flouer en vous faisant penser que tout est simple. L'obtention d'effets puissants en modifiant les valeurs de chaque pixels est très complexe. Commencer à maîtriser Numeric constitue déjà un apprentissage ardu. Dans ce tutoriel, je serai rapide avec ce qui est facile et je vais utiliser beaucoup d'exemples avec pour objectif de semer les graines de la connaissance. Après avoir fini la lecture de ce tutoriel, vous devriez comprendre les bases du fonctionnement de Surfarray.

Numeric Python

Si la paquet python Numeric n'est pas installé, il est préférable de le faire maintenant. Vous pouvez télécharger le paquet depuis cette adresse (http://sourceforge.net/project/showfiles.php?group_id=1369). Pour être certain que Numeric fonctionne chez vous, vous devriez obtenir quelque chose de ce genre à partir du mode interactif de Python.

```
>>> from Numeric import *          #Importer Numeric
>>> a = array((1,2,3,4,5))         #Créer un tableau
>>> a                              #Afficher le tableau
array([1, 2, 3, 4, 5])
>>> a[2]                           #Un index dans le tableau
3
>>> a*2                            #Un nouveau tableau avec des
valeurs doublées
array([ 2,  4,  6,  8, 10])
```

Comme vous pouvez le voir, le module Numeric nous fournit un nouveau type de données, le *array*. Cet objet contient un tableau de taille fixe, et toutes les valeurs qu'il contient sont du même type. Les tableaux peuvent aussi être multidimensionnels, et c'est de cette manière nous les utiliserons avec les images. Il y aurait un peu plus à dire à leur sujet, mais c'est suffisant pour commencer.

Si vous observez la dernière commande ci-dessus, vous verrez que les opérations mathématiques sur les tableaux du module Numeric s'appliquent à toutes les valeurs du tableau. Ce fonctionnement est appelé *elementwise operations*. Ces tableaux peuvent également être *slicés* (découpés) à la façon des listes normales. La syntaxe du découpage en slice est la même que celle utilisée avec les objets python standards *(donc révisez-la si besoin)*. Voici quelques exemples de plus sur le fonctionnement des tableaux :

```
>>> len(a)                                  #Obtenir la taille du
tableau
5
>>> a[2:]                                   #Les éléments [2] et
supérieurs
array([3, 4, 5])
>>> a[:-2]                                  #Tous exceptés les 2
derniers
array([1, 2, 3])
>>> a[2:] + a[:-2]                          #Ajout le début et la
fin
array([4, 6, 8])
>>> array((1,2,3)) + array((3,4))           #Ajout de tableau de
tailles différentes
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: frames are not aligned
```

On obtient une erreur avec la dernière commande, en essayant d'ajouter deux tableaux de tailles différentes. Pour réaliser des opérations impliquant deux tableaux (incluant les comparaisons et les assignations) les deux tableaux doivent avoir les mêmes dimensions. Il est très important de savoir que les valeurs contenues dans un tableau créé depuis le slice d'un original possède les mêmes références que les valeurs du tableau de départ. Donc modifier une valeur dans un slice issue d'un tableau original, modifiera la valeur correspondante du tableau original. Cette propriété des tableaux est très importante à retenir.

```
>>> a                    #Afficher le tableau de départ
array([1, 2, 3, 4, 5])
>>> aa = a[1:3]          #Slicer 2 éléments intermédiaires
>>> aa                   #Afficher le slice
array([2, 3])
>>> aa[1] = 13           #Modifier une valeur dans le slice
>>> a                    #Afficher le changement dans l'original
array([ 1, 2, 13,  4,  5])
>>> aaa = array(a)       #Faire une copie du tableau
>>> aaa                  #Afficher la copie
array([ 1, 2, 13,  4,  5])
>>> aaa[1:4] = 0         #Définir à 0 des valeurs intermédiaires
>>> aaa                  #Afficher la copie
array([1, 0, 0, 0, 5])
>>> a                    #Afficher l'original
array([ 1, 2, 13,  4,  5])
```

Maintenant, nous jetterons un coup d'œil à de petits tableau à deux dimensions. Ne soyez pas trop inquiets, c'est la même chose que d'avoir un tuple à deux dimensions (un tuple dans un tuple). Commençons avec des tableaux à deux dimensions.

```
>>> row1 = (1,2,3)                          #Créer un tuple
>>> row2 = (3,4,5)                          #Créer un second tuple
>>> (row1,row2)                             #Afficher comme un tuple
2D
((1, 2, 3), (3, 4, 5))
>>> b = array((row1, row2))                 #Créer un tableau 2D
>>> b                                       #Afficher le tableau
array([[1, 2, 3],
       [3, 4, 5]])
>>> array(((1,2),(3,4),(5,6)))              #Afficher un nouveau
tableau 2D
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Maintenant, avec ce tableau à deux dimensions (que l'on appellera à partir de maintenant un **2D**), nous pouvons récupérer des valeurs spécifiques par leur index et slicer dans les deux dimensions. L'utilisation d'une virgule pour séparer les indices, nous permet de chercher/slicer dans plusieurs dimensions. L'utilisation de "**:**" comme un index (afin de ne pas fournir tous les indices) nous renvoie toutes les valeurs contenues sur cette dimension. Voyons son fonctionnement :

```
>>> b                  #Afficher la tableau précédent
array([[1, 2, 3],
       [3, 4, 5]])
>>> b[0,1]             #Indexer une valeur unique
```

```
2
>>> b[1,:]              #Slicer la seconde rangée
array([3, 4, 5])
>>> b[1]                #Slicer la seconde rangée (idem ci-dessus)
array([3, 4, 5])
>>> b[:,2]              #Slicer la dernière colonne
array([3, 5])
>>> b[:,:2]             #Slicer en un tableau 2x2
array([[1, 2],
       [3, 4]])
```

Bon, restez avec moi, c'est à peu près aussi dur que ça. En utilisant Numeric, il existe une fonctionnalité supplémentaire pour effectuer des slices. Le slice de tableau nous permet de spécifier un *incrément de slice*. La syntaxe pour un slice avec incrément est `index_debut : index_fin : increment`.

```
>>> c = arange(10)                          #Comme range(), mais
pour faire un tableau
>>> c                                       #Afficher le tableau
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> c[1:6:2]                                #Slicer les valuers
impaires entre 1 et 6
array([1, 3, 5])
>>> c[4::4]                                 #Slicer toutes les 4
valeurs en démarrant à l'index 4
array([4, 8])
>>> c[8:1:-1]                               #Slice de 1 à 8
inversé
array([8, 7, 6, 5, 4, 3, 2])
```

Voilà. Vous en savez suffisamment pour vous permettre de commencer à utiliser Numeric avec le module Surfarray. Les propriétés du module Numeric sont certainement plus consistantes, mais il ne s'agit que d'une introduction. Par ailleurs, on veut seulement faire des trucs marrants, pas vrai ?

Importer le module Surfarray

Pour utiliser le module Surfarray, nous avons besoin de l'importer. Les modules Surfarray et Numeric étant des composants optionnels de Pygame, il est judicieux de s'assurer de les importer correctement avant de les utiliser. Dans ces exemples, j'importerai le module Numeric dans une variable nommée N. Vous verrez ainsi quelles fonctions utilisées provient du module Numeric (et de plus c'est légèrement plus court que de taper Numeric devant chaque fonction).

```
try:
    import Numeric as N
    import pygame.surfarray as surfarray
```

```
except ImportError:
    raise ImportError, "Numeric and Surfarray are required."
```

Introduction à Surfarray

Il y a deux principaux types de fonctions dans Surfarray. Un des jeu de fonctions concerne la création d'un tableau qui est une copie des données de pixels d'une surface. L'autre jeu de fonctions crée une copie par référence d'un tableau de pixels, donc changer le tableau modifie directement la surface originale. Il y a d'autres fonctions qui vous permettent d'accéder aux valeurs du canal alpha de chaque pixel à l'aide de tableaux et de plusieurs autres fonctions très utiles. Nous étudierons ces fonctions plus tard.

En utilisant ces tableaux de surface, il y a deux moyens de représenter les valeurs des pixels. La première, peut être de les représenter comme un carte de nombres entiers. Ce type de tableau est un simple tableau 2D avec un unique entier qui représente la couleur du pixel correspondant. Ce type de tableau est pratique pour déplacer des parties d'une image. L'autre type de tableaux utilise trois valeurs pour représenter chaque pixel en codage RGB. Ce type de tableau rend extrêmement simple la réalisation d'effets qui modifie la couleur de chaque pixel. Ce type de matrice est également un peu délicat à manipuler, puisqu'il s'agit en fait d'un tableau à 3 dimensions. Si vous parvenez malgré tout à comprendre le truc, ce n'est pas plus difficile que d'utiliser un tableau 2D normal.

Le module Numeric utilise une machine de nombres naturels pour représenter les données numériques, donc un tableau Numeric peut être constitué d'entier de 8bits, 16bits, et 32bits. (les tableaux peuvent également utiliser d'autres types comme des flottants et des doubles, mais pour notre manipulation d'image nous n'utilisons pratiquement que des entiers). Du fait de la limitation en taille de certains entiers, vous devez veiller à ce que les tableaux contenant les données des pixels soient des tableaux dont les données sont du type adéquat. Les fonctions fabriquant ces tableaux à partir de surfaces sont :

- **surfarray.pixels2d(surface)**

  Crée un tableau 2D (valeur des pixels entière) qui **référence** les données originales de la surface. Ceci fonctionnera pour tous les formats de surface excepté celles en 24 bits.

- **surfarray.array2d(surface)**

  Crée un tableau 2D (valeur des pixels entière) **copié** depuis n'importe quel type de surface.

- **surfarray.pixels3d(surface)**

  Crée un tableau 3D (valeur des pixels codé en RGB) qui **référence** les données originales d'une surface. Cela va fonctionner exclusivement avec des surfaces sur 24 bits ou 32 bits qui ont un formatage RGB et BGR.

- **surfarray.array3d(surface)**

Crée un tableau 3D (valeurs des pixels codé en RGB) **copié** depuis n'importe quel type de surface.

Voici un petit résumé qui devrait mieux illustrer quels types de fonctions devraient être utilisés sur quelles surfaces. Comme vous pouvez le voir, les fonctions de type arrayXD vont fonctionner avec tous les types de surface.

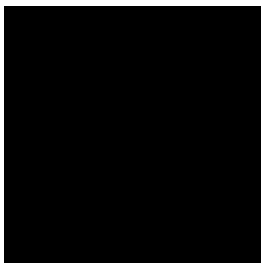
	32 bits	24 bits	16 bits	8 bits(c-map)
pixel2D	yes		yes	yes
array2D	yes	yes	yes	yes
pixel3D	yes	yes		
array3D	yes	yes	yes	yes

Exemples

Avec ces informations, nous sommes parés pour commencer à essayer diverses choses avec les tableaux de surface. Les petites démonstrations suivantes créent un tableau Numeric et l'affichent dans pygame. Ces différents tests sont issus des exemples contenus dans le fichier `arraydemo.py`. Il y a une fonction simple nommée `surfdemo_show()` qui affiche un tableau à l'écran.

Création d'une image noire

```
allblack = N.zeros((128, 128))
surfdemo_show(allblack, 'allblack')
```

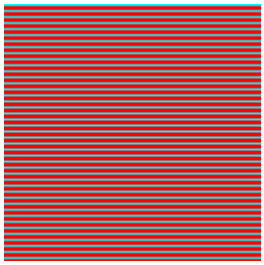


Dans notre premier exemple, nous créons un tableau entièrement noir de 128 lignes sur 128 colonnes. Pour créer un tableau numérique avec un taille déterminée, il est préférable d'utiliser la fonction `N.zeros()`. Ici, le tableau de zéros forme une surface noire.

Tableaux de 3 dimensions (séparation des composantes RGB)

```
striped = N.zeros((128, 128, 3))
striped[:] = (255, 0, 0)
striped[:,::3] = (0, 255, 255)
surfdemo_show(striped, 'striped')
```

Ici nous manipulons un tableau à 3 dimensions. On commence par créer une image rouge. Ensuite nous extrayons une ligne sur trois et nous lui donnons la couleur bleu/vert. Comme vous pouvez le constater, nous pouvons traiter les tableaux à trois dimensions presque comme un tableau à deux dimensions, seulement on lui assigne des 3-uplets au lieu de valeurs uniques (scalaires).



Extraction des données d'une image depuis un fichier

```
imgsurface = pygame.image.load('surfarray.jpg')
imgarray = surfarray.array2d(imgsurface)
surfdemo_show(imgarray, 'imgarray')
```

Ici nous chargeons une image avec la fonction `image.load()` qui la convertit en un tableau 2D d'entiers. Nous utiliserons cette image comme base dans le reste de nos exemples.

Retournement *miroir* vertical

```
flipped = imgarray[:,::-1]
surfdemo_show(flipped, 'flipped')
```

Voici un retournement vertical de l'image, réalisé en utilisant la notation en *slices* à l'aide d'un incrément négatif pour l'indice des colonnes.

Miniaturisation d'une image

```
scaledown = imgarray[::2,::2]
surfdemo_show(scaledown, 'scaledown')
```

Diminuer une image repose sur le même principe que l'exemple précédent. Ici, la notation en slices est utilisée pour conserver seulement un pixel sur deux à la fois verticalement et horizontalement.

Augmentation de la taille d'une image

```
size = N.array(imgarray.shape)*2
scaleup = N.zeros(size)
scaleup[::2,::2] = imgarray
scaleup[1::2,::2] = imgarray
scaleup[:,1::2] = scaleup[:,::2]
surfdemo_show(scaleup, 'scaleup')
```

Augmenter la taille d'une image n'est pas aussi radicalement simple, mais s'inspire de la diminution que nous avons réalisé en utilisant les slices. D'abord, nous créons un tableau qui est de deux fois la taille de l'original. On réalise une copie du tableau original, pixel par pixel, en écrivant seulement sur les colonnes paires du tableau de destination, puis on réalise à nouveau l'opération en écrivant seulement sur les colonnes impaires du tableau de destination. À ce stade, nous avons image redimensionnée correctement, mais toutes les lignes impaires sont noires. Il nous suffit alors de recopier chaque ligne paire sur la ligne du dessous. On obtient ainsi une image dont la taille a doublé.

Filtrage de canaux

```
rgbarray = surfarray.array3d(imgsurface)
redimg = N.array(rgbarray)
redimg[:,:,1:] = 0
surfdemo_show(redimg, 'redimg')
```

Retour vers les tableaux 3D, on utilisera le codage RGB pour modifier les couleurs. On fait un simple tableau 3D à partir de l'image originale, en utilisant la méthode `surfarray.array3D()`, puis toutes les valeurs pour le bleu et le vert sont mises à zéro. Il nous reste alors, uniquement le canal rouge.

Filtrage par convolution

```
soften = N.array(rgbarray)
soften[1:,:]  += rgbarray[:-1,:]*8
soften[:-1,:] += rgbarray[1:,:]*8
soften[:,1:]  += rgbarray[:,:-1]*8
soften[:,:-1] += rgbarray[:,1:]*8
soften /= 33
surfdemo_show(soften, 'soften')
```

On réalise ici une convolution à l'aide d'un filtre 3x3 qui va adoucir les reliefs de l'image. Cela paraît lourd en calculs, mais ce qui est fait est en fait de décaler l'image de 1 pixel dans toutes les directions, et de sommer toutes ces images (en multipliant par un certain coefficient de poids). Alors, on moyenne toutes les valeurs obtenues. Ce n'est pas un filtre gaussien, mais c'est rapide.

Décoloration

```
src = N.array(rgbarray)
dest = N.zeros(rgbarray.shape)
dest[:] = 20, 50, 100
diff = (dest - src) * 0.50
xfade = src + diff.astype(N.Int)
surfdemo_show(xfade, 'xfade')
```

Enfin, Nous réalisons une décoloration croisée entre l'image originale et une fond entièrement en bleu. Ce n'est pas très folichon, mais l'image de destination peut être n'importe quoi, et en modifiant le coefficient multiplicateur (0.50 dans l'exemple), vous pouvez choisir chaque étape pour un fondu linéaire entre deux images.

Conclusion

J'espère qu'à partir de maintenant vous commencez à voir comment le module Surfarray peut-être utilisé pour réaliser des effets spéciaux et/ou transformations qui ne sont possibles qu'à partir d'une manipulation de pixels. Au minimum, vous pouvez utiliser Surfarray pour faire un grand nombre d'opérations très rapides

de type `Surface.set_at()` et `Surface.get_at()`. Mais ne pensez pas que vous en ayez terminé avec ce module, il vous reste encore beaucoup à apprendre.

Verrouillage de surface

Comme le reste de Pygame, Surfarray va verrouiller tout objet de type Surface lors de l'accès aux données de pixels. C'est une chose dont il faut être conscient dans tout ce que vous faites. En créant un tableau de données de pixels, la surface originale sera verrouillée pendant le temps d'existence du tableau de donnée. Il est important de s'en rappeler. Soyez certain d'avoir supprimé le tableau de pixels soit explicitement avec l'instruction Python : `del`, soit implicitement en sortant de l'espace de nom et ainsi faire intervenir le *garbage collector* (comme par exemple après un retour de fonction).

Faites attention à ne pas accéder directement à des surfaces en hardware (*HWSURFACE*). Car les données de ces surfaces résident dans la mémoire de la carte graphique, et le transfert de modifications de pixels à travers le bus PCI/AGP n'est pas des plus rapide.

Transparence

Le module Surfarray possède plusieurs méthodes pour accéder aux valeurs du canal alpha/couleur clé d'une surface. Aucune des fonctions qui gèrent le canal alpha, n'a d'effet sur le reste des données de la surface, uniquement sur les valeurs du canal alpha des pixels. Voici la liste de ces fonctions :

- **surfarray.pixels_alpha(surface)**

  Crée un tableau 2D de valeurs entières qui **référence** les valeurs du canal alpha des pixels d'une surface. Ceci fonctionne uniquement avec les images codées sur 32 bits par pixel, avec un canal alpha sur 8 bits.

- **surfarray.array_alpha(surface)**

  Crée un tableau 2D de valeurs entières qui **copie** les valeurs du canal alpha des pixels d'une surface. Ceci fonctionne avec tous les types de surface. Si l'image d'origine ne contient aucun canal alpha, les valeurs du tableau sont initialisées à 255, qui est la valeur maximale d'opacité.

- **surfarray.array_colorkey(surface)**

  Crée un tableau 2D de valeurs entières qui met la transparence à 0 (valeur maximale de transparence) pour chaque pixel de la surface dont la couleur correspond à la couleur clé.

Autres fonctions du module Surfarray

Il existe quelques autres fonctions disponibles dans le module Surfarray. Vous pouvez en obtenir une liste exhaustive ainsi qu'une description plus complète sur la page de référence (http://pygame.seul.org/docs/ref/surfarray.html). Notez malgré tout cette fonction très utile :

- **surfarray.blit_array(surface, array)**

  Ceci va transférer tout type de tableau 2D ou 3D sur une surface possédant les mêmes dimensions. Ce blit de Surfarray sera généralement beaucoup plus rapide que d'assigner un tableau qui contiendrait les pixels de référence. Néanmoins, ça ne devrait pas être plus rapide qu'un blit normal d'une surface, puisque celui-ci est très optimisé.

Utilisation plus avancée de Numeric

Voici deux dernière choses qu'il est bon de connaître à propos des tableaux de Numeric. En manipulant des tableaux de très grande taille, comme par exemple des grandes surfaces de 640x480, vous devrez veiller à certaines choses en particulier. D'abord, même si les opérateurs + et * utilisés avec les tableaux sont très pratiques, ils sont également très coûteux en temps de calcul sur les grands tableaux. Ces opérateurs doivent réaliser des nouvelles copies temporaires des tableaux, qui sont alors habituellement copiées dans un autre tableau. Cela peut prendre énormément de temps. Heureusement, tous les opérateurs du module Numeric sont fournit avec des fonctions spéciales qui sont plus performantes et peuvent être utilisées en lieu et place des opérateurs. Par exemple, vous pourriez remplacer `screen[:] = screen + brightmap` par la fonction plus rapide `add(screen, brightmap, screen)`. Toutefois, lisez la documentation concernant les *Numeric Ufuncs* (http://numpy.scipy.org/numpydoc/numpy-7.html#pgfId-36126) pour en savoir plus à leur sujet. C'est important lors de la manipulation des tableaux.

En manipulant les tableaux avec des valeurs de pixel codés sur 16 bits, Numeric n'utilise pas des entiers non signés sur 16 bits, donc certaines de vos valeurs seront des nombres négatifs signés. Heureusement ça ne pose pas de problème.

Une autre chose à laquelle il faut veiller en utilisant des tableaux est le type de données manipulé. Certains tableaux (particulièrement les surfaces de pixels *mappées*, en codage RGB) retourne des tableaux avec des valeurs sur 8 bits non signés. Ces tableaux peuvent facilement provoquer un dépassement de capacité si vous n'êtes pas très attentifs.Le module Numeric possède les mêmes contraintes que vous trouverez dans le langage C, c'est à dire qu'une opération avec un nombre en 8 bits et un nombre en 32 bits va renvoyer un nombre en 32 bits. Vous pouvez toujours convertir le type de donnée d'un tableau, mais soyez toujours certain du type que contienne les tableaux que vous manipulez. S'il arrive une situation dans laquelle un dépassement de capacité est provoqué, Numeric va lever une exception.

Enfin, vous devez faire attention lorsque vous assignez des valeurs dans un tableau à trois dimensions, celles-ci doivent être comprises entre 0 et 255, sinon vous obtiendrez des erreurs de troncatures indéfinies.

Remise du Diplôme

Ok, vous l'avez, ma formation rapide sur Numeric python et Surfarray. Espérons que maintenant vous voyez ce qu'il est possible de faire, et que si vous ne l'avez jamais utilisé vous-même, vous ne serez pas effrayé à la vue de ces codes. Regardez dans l'exemple `vgrade.py` pour plus d'actions sur les tableaux Numeric. Il existe également quelques démonstrations "enflammées" qui utilisent Surfarray pour créer un effet de liquide en temps réel. Le mieux est toujours d'essayer des choses par vous même. Allez-y tranquillement au début, et construisez au fur et à mesure. J'ai vu des choses très intéressantes faites avec Surfarray comme des gradients radiaux et d'autres choses dans le genre. Bonne Chance.

Guide du débutant

ou ***Les choses que j'ai apprises à force d'essais et d'erreurs, et que vous n'aurez pas à reproduire***

ou ***Comment j'ai appris à arrêter de paniquer et à aimer le blit***

Traduit de l'anglais, original par *David Clark* :
http://www.pygame.org/docs/tut/newbieguide.html

Pygame est une enveloppe de la SDL (http://www.liblsd.org/) (Simply DirectMedia Layer) pour le langage Python, écrite par *Pete Shinners*. Ce qui signifie que, en utilisant Pygame, vous pouvez écrire des jeux ou d'autres applications multimédia en Python qui fonctionneront de manière identique sur toutes les plateformes supportant la SDL (Windows, Unix, Mac, beOS et autres).

Pygame peut être facile à apprendre mais le monde de la programmation graphique peut sembler déroutant pour un nouveau venu. J'ai écrit ceci pour essayer de rassembler les connaissances pratiques que j'ai acquises tout au long de l'année passée en travaillant sur Pygame et son prédecesseur pySDL. J'ai essayé de classer ces suggestions par ordre d'importance mais la pertinence de tel ou tel conseil dépendra de votre expérience personnelle et des détails de votre projet.

Règle 1 : soyez à l'aise dans votre utilisation de Python

La chose la plus importante est d'être à l'aise avec l'utilisation de Python. Apprendre quelque chose est potentiellement compliqué et la programmation graphique sera une vraie corvée si de plus, vous n'êtes pas familier avec le langage que vous utilisez.

- Écrivez quelques programmes non-graphiques en Python :
  - un parser de fichiers textes,
  - un jeu à invite de texte,
  - un programme à entrée journalière ou d'autres choses de ce style.
- Soyez à l'aise avec la manipulation de chaînes de caractères et de listes : sachez comment les découper, les slicer et combiner les listes et les chaînes.
- Apprenez comment réutiliser le travail : essayez d'écrire un programme qui utilise plusieurs fichiers sources réunis.
- Écrivez vos propres fonctions et entraînez-vous à la manipulation des nombres et des caractères,
- apprenez comment convertir les nombres en chaînes et inversement.

- Venez-en au point où la syntaxe d'utilisation des listes et des dictionnaires est une seconde nature : vous ne devez pas avoir besoin de lire la documentation à chaque fois que vous avez besoin de découper une liste ou de trier une série de clés de dictionnaire.
- Résistez à la tentation d'utiliser une mailing liste, `comp.lang.python`, irc ou un forum de discussion lorsque vous rencontrerez des problèmes.
- Au lieu de ça, saignez votre interpréteur et jouez avec le problème quelques heures.
- Imprimez le guide de référence rapide (http://rgruet.free.fr//#QuickRef) de Python et gardez-le près de votre ordinateur.

Cela peut vous paraître incroyablement ennuyeux mais l'assurance que vous aurez gagnée en vous familiarisant avec Python fonctionnera à merveille lorsque viendra le moment d'écrire votre jeu. Le temps que vous passerez à faire de Python votre seconde nature ne sera rien comparé au temps que vous gagnerez lorsque vous serez en train d'écrire du vrai code.

Règle 2 : identifiez les parties de Pygame dont vous aurez réellement besoin

Étudier le fatras des classes indiquées dans l'index de la documentation de Pygame peut être vraiment déroutant. L'important est de se rendre compte que l'on peut faire beaucoup avec un petit sous-ensemble de fonctions. Une grande partie des classes disponibles ne sera pas utilisée dans un premier temps : en un an, je n'ai pas touché aux modules Channel, Joystick, Cursors, Userrect, Surfarray ni à leurs différentes fonctions.

Règle 3 : comprenez ce qu'est une Surface

La partie la plus importante de Pygame concerne la manipulation de surfaces. Imaginez-vous qu'une surface n'est qu'un morceau de papier blanc : vous pouvez dessiner des lignes dessus, remplir certaines parties avec de la couleur, et y copier ou en extraire chaque valeur des pixels qui la constituent. Une surface peut être de n'importe quelle taille (dans la limite du raisonnable) et vous pouvez en manipuler autant que vous voulez (toujours dans la limite du raisonnable). Une seule surface est particulière : celle que vous créez avec la fonction `pygame.display.set_mode()`. Cette *surface d'affichage* représente l'écran : ce que vous y faites apparaîtra sur l'écran de l'utilisateur. Vous ne pouvez en avoir qu'une seule à la fois : c'est une limitation de la SDL, pas de Pygame.

Donc, comment créer des surfaces ? Comme mentionné ci-dessus, vous créez la surface spéciale *surface d'affichage* avec `pygame.display.set_mode()`. Vous pouvez créer une surface qui contient une image en utilisant `image.load()` ou du texte avec `font.render()`. Vous pouvez également créer une surface qui ne contient rien du tout avec `Surface()`.

La plupart des fonctions de manipulation de surface ne sont pas d'une utilité critique. Apprenez seulement `blit()`, `fill()`, `set_at()` et `get_at()` et tout ira bien.

Règle 4 : utilisez *surface.convert()*

Quand j'ai commencé à lire la documentation de `surface.convert()`, je pensais ne pas en avoir besoin car j'utilisais exclusivement le format PNG pour ne pas avoir de problème de format d'image ; je n'avais pas besoin de `convert()`. J'ai réalisé que j'avais vraiment, vraiment tort.

Le *format* auquel `convert()` fait référence n'est pas un *format* de fichier (comme PNG, JPEG, GIF), c'est ce qui s'appelle l'espace colorimétrique (RGB/HSV/YUV/...). Cela se réfère à la façon particulière qu'a une surface, d'enregistrer les différentes couleurs dans un pixel spécifique. Si le *format* de la surface n'est pas le même que le *format* d'affichage, SDL devra convertir à la volée chaque blit, ce qui est très coûteux en temps de calcul. Ne vous souciez pas plus que ça des explications : souvenez-vous seulement que `convert()` est nécessaire si vous ne voulez pas que votre affichage soit ralenti inutilement.

Comment devez-vous utiliser `convert()` ? Appelez-la après avoir créé une surface avec la fonction `image.load()`. Au lieu de faire :

```
surface = pygame.image.load('foo.png')
```

Privilégiez :

```
surface = pygame.image.load('foo.png').convert()
```

C'est simple, vous avez besoin de ne l'appeler qu'une seule fois par surface, lorsque vous chargez votre image depuis le disque et vous serez enchanté des résultats. J'ai remarqué un gain de performance sur les blits de l'ordre de 6x en utilisant la fonction `convert()`.

Les seules fois où vous ne voudrez pas utiliser la fonction `convert()`, est lorsque vous avez absolument besoin de garder un contrôle absolu sur le format interne de l'image - comme par exemple lorsque vous écrivez un logiciel de conversion d'image ou s'en approchant et que vous devez vous assurer que le fichier de sortie possède le même espace colorimétrique que le fichier d'entrée. Si vous écrivez un jeu, vous avez besoin de vitesse, donc utilisez la fonction `convert()`.

Règle 5 : l'animation par *dirty_rect*

La cause principale d'un taux d'images inadéquate dans un programme Pygame résulte d'un malentendu sur la fonction `pygame.display.update()`. Avec Pygame, le seul tracé sur la *surface d'affichage* n'engendre pas son apparition sur l'écran : vous avez besoin d'appeler la fonction `pygame.display.update()`. Il existe trois manières d'appeler cette fonction :

pygame.display.update()
: Celle-ci actualise la fenêtre entière (ou l'écran entier lors d'un affichage en plein écran).

pygame.display.flip()
: Celle-là fait la même chose mais devrait être celle utilisée si vous utilisez l'accélération matérielle en double tampon (doublebuffer), que vous n'avez pas, ainsi sur...

pygame.display.update (un rectangle ou une liste de rectangles)
: Cette dernière actualise uniquement les zones rectangulaires de l'écran que vous avez spécifiées.

La plupart des personnes débutantes en programmation graphique utilisent la première : ils mettent à jour la totalité de l'écran à chaque image. Le problème est que c'est inacceptablement lent pour la plupart des personnes. Appeler `update()` prends 35 millisecondes sur ma machine, ce qui n'est pas énorme, jusqu'à ce que vous réalisiez que 1000 ms / 3 5ms = 28 images par secondes maximum. Et ceci sans la logique de jeu, sans blits, sans entrées, sans intelligence artificielle, sans rien. Je n'ai simplement fait qu'actualiser l'écran et 28 images par secondes est mon taux maximal. Hum...

La solution est appelée *dirty rect animation*. Au lieu d'actualiser l'écran entier à chaque image, seule la partie qui a changé depuis la dernière image est actualisée. J'obtiens ceci en suivant ces rectangles dans une liste, ensuite j'appelle `update(the_dirty_rectangles)` à la fin de l'image. En détail, pour le déplacement d'un sprite, je :

1. Blit une partie de l'arrière-plan sur l'emplacement actuel du sprite, ce qui l'efface.
2. Ajoute le rectangle de l'emplacement actuel du sprite dans une liste appelée `dirty_rects[]`.
3. Déplace le sprite.
4. Dessine le sprite sur son nouvel emplacement.
5. Ajoute le nouvel emplacement du sprite sur ma liste de `dirty_rects`.
6. Appelle la fonction `display.update(dirty_rects)`.

La différence de vitesse est stupéfiante. En considérant que Solarwolf (http://www.pygame.org/shredwheat/solarwolf/) possède des douzaines de sprites en mouvement mis à jour de façon fluide et qu'il lui reste encore assez de temps pour afficher un champ d'étoiles en *parallax* en arrière-plan et l'actualiser lui aussi.

Il existe deux cas où cette technique ne fonctionne pas. Le premier est lorsque la fenêtre ou l'écran entier doit être actualisé entièrement : pensez à un moteur de scrolling fluide comme une vue aérienne de jeu de stratégie en temps réel ou un jeu à défilement latéral. Alors que faites-vous dans ces cas-là ? Voici la réponse courte : *N'écrivez pas ce genre de jeu avec Pygame*. La réponse longue est de faire défiler par étapes une grosse quantité de pixels à la fois. N'essayez pas d'obtenir un scrolling parfaitement fluide. Vos joueurs apprécieront un jeu qui défile rapidement et ne vous tiendront pas trop rigueur sur les sauts de l'arrière-plan.

Un dernier mot : tous les jeux ne requièrent pas de fort taux de rafraichissement. Un jeu de stratégie, de style wargame, pourrait facilement s'accommoder de quelques images par secondes ; dans ce cas, la complexité ajoutée par l'animation en *dirty_rect* ne serait pas nécessaire.

Règle 6 : les surfaces matérielles engendrent plus de problèmes que d'avantages

Si vous avez étudié les différents drapeaux utilisables dans la fonction `pygame.display.set_mode()`, vous pouvez vous dire: *"Aaah, HWSURFACE ! Cool, c'est ce dont j'ai besoin, qui n'utilise pas l'accélération matérielle ? Oooh DOUBLEBUF ! Ca m'a l'air rapide, je pense que je vais l'utiliser aussi !"*. Ce n'est pas votre faute, nous avons été habitués, par les jeux 3D, à croire que l'accélération matérielle est meilleure et que le rendu logiciel est lent.

Malheureusement, l'accélération matérielle engendre une longue liste d'inconvénients :

- Elle ne fonctionne que sur certaines plateformes. Les machines Windows peuvent habituellement accéder aux surfaces matérielles si vous les demandez. La plupart des autres plateformes ne le peuvent pas. Linux, par exemple, est capable de fournir des surfaces matérielles si Xorg4 est installé, si DGA2 fonctionne correctement et que les lunes sont correctement alignées (*NdT : Ce guide doit dater un peu, ça fait quelques temps que les pilotes sont devenus potables. Don't feed the troll ;)*). Si les surfaces matérielles ne sont pas disponibles, la SDL vous fournira une surface logicielle à la place.
- Elle fonctionne uniquement en plein écran.
- Elle complique les accès aux pixels. Si vous avez une surface matérielle, vous avez besoin de la verrouiller avant d'écrire ou de lire la valeur d'un seul pixel de celle-ci. Si

vous ne le faites pas, de *mauvaises choses arriveront*. Alors vous devrez rapidement déverrouiller la surface, avant que l'OS s'embrouille et commence à paniquer. La plupart de ces processus sont automatisés par Pygame, mais ce sont des éléments à prendre en compte.

- Vous perdez le pointeur de la souris. Si vous spécifiez HWSURFACE (et que vous l'obtenez), votre pointeur va simplement s'évaporer (ou pire, s'accrocher à droite ou à gauche et commencer à scintiller). Vous aurez besoin de créer un sprite pour le pointeur de la souris, et vous aurez besoin de faire attention à l'accélération du pointeur et à sa sensibilité. Que de complications...
- Toutefois, ce pourra toujours être lent. La plupart des pilotes ne sont pas accélérés pour le type de tracé que nous faisons et puisque que tout doit être blité à travers le bus vidéo (à moins que vous ne puissiez fourrer votre surface source dans la mémoire vidéo aussi), ça pourra finir par être aussi lent qu'un accès logiciel.

Le rendu matériel garde son utilité. Il fonctionne de manière fiable sous Windows ; si vous n'êtes pas intéressé par des performances multi-plateforme, il peut vous fournir une augmentation de vitesse substantielle. Cependant, il a un coût : augmenter les maux de têtes et la complexité. Il est préférable de conserver les bonnes vieilles SWSURFACES fiables, jusqu'à ce que vous soyez certains de ce que vous faites.

Règle 7 : ne soyez pas distrait par des questions secondaires

Parfois, les programmeurs de jeux débutants passent énormément de temps à se soucier de questions qui ne sont pas vraiment critiques pour le succès de leur jeu. Le désir de satisfaire des objectifs secondaires est compréhensible mais au début du processus de la création d'un jeu, vous ne pouvez pas savoir quelles sont les questions importantes, sans parler des réponses que vous devrez choisir. Le résultat peut engendrer de nombreuses tergiversations inutiles.

Par exemple, considérons la question : *comment organiser les fichiers de vos graphismes*. Est-ce que chaque image devrait avoir son propre fichier ? ou chaque sprite ? Peut-être que tous les graphismes devraient être zippés dans une archive ? Énormément de temps a été perdu pour beaucoup de projets en posant ces questions sur des listes de diffusion, en débattant des réponses, en peaufinant, etc. Ce ne sont que des questions secondaires, chaque instant passé à discuter devrait être passé à coder le jeu.

En résumé, il est de loin préférable de mettre en œuvre une assez bonne solution avec succès, plutôt que de tenter en vain une solution parfaite que l'on ne sait pas comment coder.

Règle 8 : les Rects sont vos amis

L'enveloppe de Pete Shinner (Pygame) peut fournir de beaux effets de transparence et de bonnes vitesse de blit mais je dois admettre que ma partie préférée de Pygame est la modeste classe `Rect`. Un `rect` est un simple rectangle, défini par la position de son coin supérieur gauche, sa largeur et sa hauteur. Beaucoup de fonctions de Pygame prennent des `rects` en arguments ou des styles de `rects`, ou encore des séquences qui ont les mêmes valeurs qu'un `rect`. Ainsi, si je veux un rectangle qui définit une zone entre 10, 20 et 40, 50, je peux faire une des choses suivantes :

```
rect = pygame.Rect(10, 20, 30, 30)
rect = pygame.Rect((10, 20, 30, 30))
rect = pygame.Rect((10, 20), (30, 30))
```

```
rect = (10, 20, 30, 30)
rect = ((10, 20, 30, 30))
```

Si vous utilisez une des trois premières versions, quelle qu'elle soit, vous aurez accès aux fonctions utilitaires des `Rects`. Elles incluent les fonctions de déplacement, de diminution et d'agrandissement des `rects`, de recherche de l'union de deux `rects`, et d'une variété de fonctions de détection de collision.

Par exemple, je suppose que j'aimerais obtenir une liste de tous les sprites qui contiennent le point `(x, y)`, peut-être que le joueur a cliqué ici, ou peut-être est-ce l'emplacement actuel d'une balle. C'est très simple si chaque sprite possède un attribut `rect`, je n'ai qu'à faire :

```
sprites_clicked = [sprite for sprite in
toute_ma_liste_de_sprites if sprite.rect.collidepoint(x, y)]
```

Les `Rects` n'ont avec les surfaces ou les fonctions graphiques aucune autre relation que le fait de les utiliser comme arguments. Vous pouvez les utiliser à des endroits qui n'ont rien à voir avec le graphisme mais que vous avez besoin de définir comme des rectangles. À chaque projet, je découvre de nouvelles façons d'utiliser des `rects`, là où je n'avais jamais pensé en avoir besoin.

Règle 9 : ne vous tracassez pas avec une détection de collision au pixel près

Vous avez donc vos sprites qui se déplacent et vous avez besoin de savoir s'ils entrent en collision ou non. Vous pouvez tenter d'écrire quelque chose comme ceci :

1. Vérifier si les rects entrent en collision. Sinon, les ignorer.
2. Pour chaque pixel qui se chevauche avec un autre, voir si les pixels correspondant des deux sprites sont opaques. Si oui, il y a collision.

Il existe d'autres solutions, en ajoutant des masques de sprites, mais comme vous devez le faire dans Pygame, ce sera probablement trop lent. Pour la plupart des jeux, il sera préférable de tester une collision de *sous-rect* : en créant un `rect` pour chaque sprite qui sera un peu plus petit que l'image actuelle et l'utiliser pour les collisions. Ce sera bien plus rapide et dans la plupart des cas, le joueur ne vous tiendra pas rigueur de l'imprécision.

Règle 10 : gestion du sous-système d'évènements

Le système d'évènements de Pygame est quelque peu complexe. Il existe en fait deux manières différentes de savoir ce que fait un périphérique d'entrée (clavier, souris, joystick).

Le premier est de contrôler directement l'état du périphérique. Vous réalisez ceci en appelant `pygame.mouse.get_pos()` ou `pygame.key.get_pressed()`. Ceci vous donnera l'état du périphérique au moment de l'appel de la fonction.

La seconde méthode utilise la file d'évènement de la SDL. Cette file est une liste d'évènements : les évènements sont ajoutés à la suite de la file lorsqu'ils sont détectés et ils sont effacés de la file lorsqu'ils ont été consultés.

Il existe des avantages et des inconvénients pour chaque système. Le contrôle d'état (système 1) vous donne la précision : vous savez exactement quelle entrée a été effectuée ; si `mouse.get_pressed([0])` est vrai, cela signifie que le bouton gauche de la souris est actuellement enfoncé. La file d'évènements, elle, ne fait que rapporter que le bouton de la souris a été enfoncé à un certain moment dans le passé. Si vous vérifiez la file relativement souvent, ça fonctionnera, mais si vous tardez à la consulter, la latence peut s'agrandir. Un autre avantage du système de contrôle d'état est qu'il détecte facilement les *accords* de touches, qui sont plusieurs états au même moment. Si vous voulez savoir si les touches T et F sont pressé en même temps, il suffit de vérifier :

```
if (key.get_pressed[K_t] and key.get_pressed[K_f]):
     print "Yup!"
```

Toutefois, dans le système de file, chaque pression de touche entre dans la file comme un évènement complètement séparé ; ainsi, vous devez vous rappeler que la touche T est enfoncée et n'a pas encore été relâchée lorsque vous contrôlez l'état de la touche F . Un peu plus complexe.

Le système d'état possède toutefois une grande faiblesse. Il rapporte seulement quel est l'état d'un périphérique au moment où il est appelé. Si l'utilisateur enfonce le bouton de la souris et qu'il le relâche juste avant que l'appel à `mouse.get_pressed()` soit fait, le bouton de la souris retournera 0. La fonction `get_pressed()` rate complètement la pression du bouton de la souris. Les deux évènements, `MOUSEBUTTONDOWN` et `MOUSEBUTTONUP` seront toutefois toujours dans la file d'évènements, attendant d'être retrouvés et mis en application.

La leçon à retenir est : choisissez le système qui convient à vos besoins. Si vous n'avez pas beaucoup de continuité dans votre boucle, c'est-à-dire que vous attendez une entrée, dans une boucle `while 1:`, utilisez la fonction `get_pressed()` ou une autre fonction d'état, la latence sera réduite. D'un autre côté, si toutes les touches enfoncées sont cruciales, mais que la latence n'est pas importante, comme par exemple si l'utilisateur est en train d'écrire quelque chose dans une boite d'édition, utilisez la file d'évènements. Certaines pressions de touches pourront être un peu en retard mais au final, vous les aurez toutes.

Un mot à propos de la différence entre les fonctions `event.poll()` et `event.wait()` :

- `poll()` peut sembler meilleure puisqu'elle n'interdit pas votre programme de faire autre chose que d'attendre une entrée.
- `wait()` suspend la programme jusqu'à ce qu'un évènement soit reçu.

Toutefois, `poll()` utilisera 100% de la charge du processeur lors de son fonctionnement et il remplira la file d'évènements avec des `NOEVENTS`. Préférer l'utilisation de la fonction `set_blocked()` pour sélectionner uniquement les types d'évènements qui vous intéressent, votre file n'en sera que plus gérable.

Règle 11 : *Couleur Clé* contre *Transparence Alpha*

Il y existe de nombreuses confusions autour de ces deux techniques et beaucoup proviennent de la terminologie utilisée.

Le blit par *Couleur Clé* implique de dire à Pygame que, dans une certaine image, tous les pixels d'une certaine couleur (la *Couleur Clé* en question) apparaîtront comme transparents au lieu de s'afficher dans leur vraie couleur. C'est de cette façon que l'on crée un sprite qui n'apparaît pas dans un rectangle. Il suffit d'appeler la fonction `surface.set_colorkey(color)`, où `color` est un 3-uplets RGB, comme par exemple (0,0,0). Ceci fera que tous les pixels noirs de l'image source apparaîtront comme transparents.

La *Transparence Alpha* est différente et implique deux gestions différentes. *Image Alpha* s'applique à toute l'image et correspond probablement à ce que vous désirez. Connu aussi sous le nom de *translucidité*, le canal alpha applique à chaque pixel de l'image source une opacité partielle. Par exemple, si vous définissez le canal alpha d'une surface à 192, et que vous le blitez sur un arrière-plan, 3/4 de la couleur de chaque pixel proviendra de l'image source et 1/4 de l'arrière-plan. Le canal alpha se mesure de 255 à 0, où 0 est complètement transparent et 255 est complètement opaque. À noter que la *Couleur Clé* et le blit *Transparence Alpha* peuvent être combinés : cela produit une image complètement transparente sur certains pixels et semi-transparente sur d'autres.

La *Transparence Alpha par Pixel* est la seconde gestion du canal alpha, elle est plus complexe. Concrètement, chaque pixel d'une image source possède sa propre valeur de canal alpha, de 0 à 255. Chaque pixel peut donc avoir une opacité spécifique lorsqu'il est blités sur un arrière-plan. Ce type d'alpha ne peut pas se combiner avec une *couleur clé* et il désactive l'autre gestion de la *Transparence Alpha*. La *Transparence Alpha par Pixel* est rarement utilisée dans les jeux et pour l'utiliser vous devez enregistrer vos images sources à l'aide d'un éditeur graphique qui gère le canal alpha. C'est compliqué, ne l'utilisez pas pour l'instant.

Règle 12 : faites les choses de manière Pythonique

Un dernier mot (ce n'est pas le moins important, c'est seulement le dernier). Pygame est une enveloppe plutôt légère de la SDL, qui elle-même est une enveloppe plutôt légère des appels graphiques de votre OS. Si votre code est encore lent et que vous avez appliqué les choses que j'ai mentionnée plus haut, il y a de fortes chances que le problème vienne de la façon dont vous avez adressé vos données en Python. En Python, certains idiomes resteront lents, quoi que vous fassiez. Heureusement, Python est un langage très clair - Si une partie du code vous semble maladroite ou difficile à manier, il y a de fortes chances qu'elle puisse être optimisée en vitesse. Lisez Python Performance Tips (http://www.szgti.bmf.hu/harp/python/fastpython.html) pour trouver de précieux conseils sur la façon dont vous pouvez augmenter la vitesse de votre code. Ceci dit, une optimisation prématurée est foncièrement mauvaise, si ce n'est pas assez rapide, ne torturez pas le code pour l'accélérer. Certaines choses ne sont pas censées l'être :)

Alors voilà, maintenant vous en savez pratiquement autant que moi sur l'utilisation Pygame, allez donc écrire votre jeu !

David Clark est un utilisateur avide de Pygame et l'éditeur du Pygame Code Repository (http://www.pygame.org/pcr/), une vitrine de codes de jeu Python soumis à la communauté. C'est également l'auteur de Twitch (http://www.pygame.org/projects/20/48/), un jeu d'arcade entièrement fait avec Pygame.

Concevoir des jeux avec Pygame

Introduction

En premier lieu, je supposerais que vous ayez lu le tutoriel Chimp - Ligne par ligne, lequel introduit les bases de Python et de Pygame. Dans le cas contraire, prenez-en connaissance avant de lire la suite, car je ne répéterais pas les bases fournies par cet autre tutoriel (en tous cas, pas dans les détails). Ce tutoriel est destiné à ceux qui savent réaliser un petit jeu *ridiculement* simple, et qui aimeraient réaliser un petit jeu *relativement* simple comme Pong. Il vous fournira une introduction à quelques concepts, comme l'architecture d'un jeu, quelques notions de mathématiques pour le fonctionnement physique de la balle, ainsi que sur la manière de garder votre jeu facile à maintenir et à améliorer.

Tout le code de ce tutoriel est utilisé dans *Tom's Pong*, un jeu que j'ai écrit. À la fin de ce tutoriel, vous devriez non seulement renforcer votre compréhension de Pygame, mais vous devriez aussi comprendre comment Tom's Pong fonctionne, et comment concevoir votre propre version.

Maintenant, faisons une brève revue des bases sur l'utilisation de Pygame. Une méthode répandue d'organisation de code pour un jeu est de le diviser en six parties distinctes :

- **Le chargement des modules qui sont utilisés dans le jeu** : Tous les modules standards, excepté les importations locales des espaces de nommage de Pygame et le module Pygame lui-même.
- **Les classes de manipulation des ressources** : La définition des classes gérant la plupart des ressources de base, que sont le chargement des images et des sons, ainsi que les procédures de connexion/déconnexion au réseau, le chargement des sauvegardes de jeu, et toutes les autres ressources que vous pouvez utiliser dans votre jeu.
- **Les classes des objets du jeu** : Cette partie devra contenir les définitions de classes pour les objets de votre jeu. Dans l'exemple de Pong, ce sera un objet pour la raquette du joueur (que vous pouvez initialiser plusieurs fois, une pour chaque joueur dans le jeu), et une pour la balle (laquelle peut aussi avoir de multiples instances). Si vous souhaitez avoir un menu sympa pour votre jeu, c'est aussi une bonne idée de faire une classe pour le menu.
- **Toutes les autres fonctions du jeu** : Dans cette partie seront contenues toutes les autres fonctions nécessaires à la bonne marche du jeu, comme par exemple celles qui définissent le tableau de scores, la gestion du menu, etc. Tout le code que vous pourriez mettre dans la logique de jeu principal, mais qui rendrait sa logique difficilement lisible et peu cohérente, devrait être contenu dans cette partie. Par exemple, tracer le tableau de scores ne relève pas du jeu en lui-même, cela devrait être fait par une fonction particulière située dans cette partie.

- **L'initialisation du jeu** : Cela inclut les objets Pygame eux-mêmes, l'arrière-plan, les objets du jeu (initialisation des instances de classe) et les autres petits morceaux de code que vous pourriez vouloir ajouter.
- **La boucle principale** : C'est dans cette boucle que vous placerez la gestion des entrées (c'est à dire l'acquisition des évènements utilisateurs que sont les frappes de clavier/bouton de souris), le code de mise à jour des objets du jeu, et finalement la mise à jour de l'écran.

Tous les jeux que vous ferez auront certaines, voire la totalité de ces sections, et probablement d'autres de votre propre cru. Dans le cadre de ce tutoriel, je parlerai de la façon dont Tom's Pong est agencé, et de la façon d'appliquer cette organisation à chaque projet de jeu que vous pourriez avoir. Je supposerai également que vous voudriez garder tout le code dans un seul fichier, mais si vous faites un jeu plutôt conséquent en taille de code, c'est souvent une bonne idée de séparer le jeu en plusieurs modules. Mettre les classes des objets du jeu dans un fichier `objects.py`, par exemple, peut vous aider à séparer la logique du jeu de ses objets. Si vous avez énormément de code pour la manipulation des ressources, il peut également être pratique de le mettre dans un module `ressources.py`. Vous pourrez alors écrire `from objects, resources import *` pour importer toutes les classes et les fonctions.

Une remarque sur les styles d'écriture

La première chose à laquelle il faut penser, lors de l'approche d'un projet de programmation, est de décider d'un style d'écriture, et de le conserver. Python en lui-même facilite cela, à cause de son interprétation stricte des espaces et de l'indentation, mais cela ne vous empêche pas de choisir la largeur de votre indentation, de quelle manière vous placerez les importations, comment vous allez commenter le code, etc. Vous verrez comment je fais tout cela dans mes exemples de code, mais quel que soit le style que vous adopterez, conservez-le tout au long de votre code. Essayez également de documenter toutes vos classes, et de commenter tous les morceaux de code qui peuvent sembler obscurs. Par ailleurs, il ne sert à rien de commenter ce qui est évident. J'ai vu beaucoup de personnes faire la chose suivante :

```
player1.score += scoreup    # Add scoreup to player1 score
```

Ce n'est pas très grave, mais un peu inutile. Un mauvais code est mal agencé, avec des changements aléatoires dans le style d'écriture, et une maigre documentation. Ce mauvais code ne sera pas seulement ennuyeux pour les autres personnes, mais il sera également difficile à maintenir pour vous.

Révision : les fondamentaux de Pygame

Le jeu Pygame de base

Pour la révision (ça ne peut pas faire de mal), et pour s'assurer que vous êtes familier avec la structure d'un programme Pygame standard, je vais brièvement parcourir un programme Pygame simple, qui n'affichera rien de plus qu'une fenêtre avec un peu de texte à l'intérieur, et qui devrait finalement ressembler à ça (naturellement, la décoration de la fenêtre pourra être différente sur votre système) :

Dans cet exemple, le code complet ressemble à ça :

```
#!/usr/bin/python
# coding: utf-8

import pygame
from pygame.locals import *

def main():
    # Initialisation de la fenêtre d'affichage
    pygame.init()
    screen = pygame.display.set_mode((300, 50))
    pygame.display.set_caption('Programme Pygame de base')

    # Remplissage de l'arrière-plan
    background = pygame.Surface(screen.get_size())
    background = background.convert()
    background.fill((250, 250, 250))

    # Affichage d'un texte
    font = pygame.font.Font(None, 36)
    text = font.render("Salut tout le monde", 1, (10, 10, 10))
    textpos = text.get_rect()
    textpos.centerx = background.get_rect().centerx
    textpos.centery = background.get_rect().centery
    background.blit(text, textpos)

    # Transférer le tout dans la fenêtre
    screen.blit(background, (0, 0))
    pygame.display.flip()

    # Boucle d'évènements
    while 1:
        for event in pygame.event.get():
            if event.type == QUIT:
                return

        screen.blit(background, (0, 0))
        pygame.display.flip()
```

```
if __name__ == '__main__': main()
```

Objets Pygame de base

Comme vous pouvez le constater, le code se divise en trois catégories principales : la fenêtre d'affichage (`screen`), l'arrière-plan (`background`) et le texte (`text`). Chacun de ces objets a pu être créé grâce à l'appel en premier lieu de la méthode `pygame.init()`, que nous avons modifiée ensuite pour qu'elle convienne à nos besoins. La fenêtre d'affichage est un cas un peu spécial, car elle modifie l'affichage à travers les appels `pygame`, plutôt que d'appeler les méthodes appartenant aux objets de l'écran. Mais pour tous les autres objets Pygame, nous créons d'abord l'objet comme une copie d'un objet Pygame, en lui affectant certains attributs, et développons les objets de notre jeu à partir de celui-ci.

Pour l'arrière-plan, nous créons d'abord un objet Surface et lui donnons la taille de la fenêtre. Nous utilisons ensuite la méthode `convert()` pour convertir la Surface en un unique espace colorimétrique. C'est particulièrement recommandé lorsque nous manipulons plusieurs images et surfaces, toutes dans un espace colorimétrique différent, sinon cela ralentirait de beaucoup le rendu. En convertissant toutes les surfaces, nous pouvons accélérer drastiquement les temps de rendu. Enfin nous remplissons la surface d'arrière-plan en blanc (255, 255, 255). Ces valeurs sont en RGB et nous pouvons les retrouver à partir de n'importe quel bon programme de dessin.

En ce qui concerne le texte, nous avons besoin de plus d'un objet. D'abord nous créons un objet `font`, qui définira quelle police nous utiliserons, ainsi que sa taille. Ensuite nous créons un objet `text`, en utilisant la méthode de rendu de notre objet `font` et en lui fournissant trois arguments : le texte à faire le rendu, qui sera ou non anti-crénelé (1=oui, 0= non), ainsi que la couleur du texte (toujours dans un format RGB). Ensuite nous créons un troisième objet texte qui fournira le rectangle du texte. La manière la plus simple à comprendre est de s'imaginer en train de dessiner un rectangle qui englobera tout le texte. Vous pourrez alors utiliser ce rectangle afin d'obtenir ou de définir la position du texte sur la fenêtre d'affichage. Ainsi dans cet exemple nous avons le rectangle, et définissons ses attributs `centerx` et `centery` pour correspondre aux `centerx` et `centery` de l'arrière-plan, alors le texte aura le même centre que l'arrière-plan. Dans cet exemple, le texte sera centré sur les axes `x` et `y` de la fenêtre d'affichage.

Transfert pour afficher

Maintenant que nous avons créé les objets de notre jeu, nous avons besoin d'en faire le rendu. Si nous ne le faisons pas et que nous exécutons le programme, nous ne verrons qu'une fenêtre blanche, et nos objets resteront invisibles. Le terme employé pour faire un rendu des objets est le *blitting* (*blit* contraction du nom de la fonction BitBlT signifiant Transfert d'un bloc de bits), qui correspond à la copie de pixels d'un objet source vers un objet de destination. Ainsi pour faire une rendu de l'objet `background`, vous le transférez sur l'objet `screen`. Dans cet exemple, pour faire les choses simples, nous transférons le texte sur l'arrière-plan (donc l'arrière-plan possède une copie du texte sur lui), et ensuite nous transférons l'arrière-plan sur l'écran.

Le transfert est une des opérations les plus lentes dans un jeu, vous devez donc faire attention à ne pas trop faire de transferts sur l'écran pour chaque image. Par exemple, si vous avez une image d'arrière-plan, et une balle se déplaçant à travers l'écran, alors vous pouvez transférer l'arrière-plan en entier et ensuite la balle, tout ceci à chaque image, ce qui recouvrira la position précédente de la balle et fera un rendu de la nouvelle

balle, mais ce sera plutôt lent. Une meilleure solution consiste à transférer une partie de l'arrière-plan sur la zone occupée par la balle à l'image précédente, qui peut être trouvée grâce au `rect` de la balle précédente, et ensuite afficher la nouvelle balle, ce qui aura pour effet de transférer seulement deux petites zones.

La boucle d'évènement

Une fois que vous avez défini le jeu, vous avez besoin de le mettre dans une boucle qui s'exécutera en continu jusqu'à ce que l'utilisateur signale qu'il veuille quitter. Vous démarrerez donc une boucle ouverte, et à chaque itération de la boucle, qui sera chaque image du jeu, vous actualiserez le jeu. La première chose à contrôler pour chaque évènement, est de savoir si l'utilisateur à enfoncé une touche du clavier, cliqué un bouton de la souris, déplacé le joystick, redimensionné la fenêtre, ou tenté de la fermer. Dans ce cas, nous voudrons simplement examiner si l'utilisateur a essayé de fermer la fenêtre, auquel cas le jeu engendrera un `return`, ce qui terminera la boucle `while`. Alors nous aurons simplement besoin de re-transférer l'arrière-plan, et faire un *flip* (actualisation de l'affichage par changement de la zone mémoire affichée) de l'écran pour que chaque chose soit redessinée. D'accord, étant donné que rien ne se passe ou se déplace dans cet exemple, nous n'avons aucunement besoin de re-transférer l'arrière-plan à chaque itération, mais je le met parce que si certaines choses se déplacent à travers l'écran, vous aurez besoin de faire tous vos transferts ici.

Ta-da !

Et voilà, votre jeu Pygame le plus basique. Tous les jeux prendront une forme similaire, mais avec beaucoup plus de code concernant les fonctions de jeu elles-mêmes, que vous concevrez vous même sans les copier depuis un tutoriel ou un guide. C'est le but principal de ce tutoriel, nous allons maintenant rentrer dans le vif du sujet de la conception de jeux vidéo.

Coup d'envoi

Les premières sections du code sont relativement simples, et une fois écrites peuvent souvent être réutilisées dans d'autres jeux que vous programmerez. Elles s'occuperont de toutes les tâches fastidieuses et génériques comme : charger des modules, charger des images, ouvrir des connections réseau, jouer de la musique, etc. Elles incluront également de simples mais efficaces gestionnaire d'erreurs, et les quelques personnalisations que vous souhaiterez effectuer par dessus les fonctions fournies par des modules comme `sys` et `pygame`.

Les premières lignes et le chargement de modules

Tout d'abord, vous avez besoin de démarrer votre jeu et de charger vos modules. C'est toujours une bonne idée de définir certaines choses directement en haut du fichier source principal, comme : le nom du fichier, ce qu'il contient, sa licence, ainsi que n'importe quelle autre information que vous jugerez utile de faire lire à ceux qui la regardent. Ensuite vous pouvez charger des modules, agrémentés d'une gestion d'erreur qui fera en sorte que Python ne vous affichera pas ces horribles *traceback* que les non-programmeurs ne comprennent pas. Le code est très simple, je ne m'étendrai pas dessus :

```
#!/usr/bin/env python
# coding: utf-8
#
# Tom's Pong
```

```
# A simple pong game with realistic physics and AI
# http://tom.acrewoods.net/projects/pong
#
# Released under the GNU General Public License

VERSION = "0.4"

try:
    import sys
    import random
    import math
    import os
    import getopt
    import pygame
    from socket import *
    from pygame.locals import *
except ImportError, err:
    print "Impossible de charger le module. %s" % (err)
    sys.exit(2)
```

Fonctions de gestion des ressources

Dans l'exemple Chimp - Ligne par ligne, le premier code à être écrit correspond au chargement des images et des sons. Étant donné que c'est totalement indépendant de la logique de jeu et/ou des objets de jeu, elles seront écrites en premier et dans des fonctions séparées, ce qui impliquera que le code qui s'ensuivra pourra les utiliser. Je mets généralement tout mon code de cette nature au départ, dans leur propre fonctions, sans classe. Cela correspondra aux fonctions de gestions des ressources. Vous pouvez bien sûr créer des classes pour celle-ci, c'est à vous de développer votre propre style et vos meilleures pratiques.

C'est toujours une bonne idée d'écrire vos propres fonctions de gestion de ressources, car bien que Pygame possède des méthodes pour l'ouverture des images et des sons (ainsi que d'autres modules qui possèdent eux aussi leurs propres méthodes pour l'ouverture d'autres ressources), ces méthodes peuvent prendre plus d'une ligne, et peuvent requérir de consistantes modifications faites par vous-mêmes, et bien souvent elles ne fournissent pas de gestion d'erreur satisfaisante. Écrire des fonction de gestion de ressources vous donne un code sophistiqué, réutilisable, et vous offre plus de contrôle sur vos ressources. Prenez cet exemple d'une fonction de chargement d'image :

```
def load_png(name):
    """Charge une image et retourne un objet image"""
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
        if image.get_alpha() is None:
            image = image.convert()
        else:
            image = image.convert_alpha()
    except pygame.error, message:
            print "Impossible de charger l'image : ", fullname
```

```
            raise SystemExit, message
    return image, image.get_rect()
```

Ici nous avons créé une fonction de chargement d'image plus sophistiquée que celle fournie par Pygame : `image.load()`. À noter que la première ligne de la fonction débute par un *docstring* (chaîne de caractère de documentation) qui décrit ce que fait la fonction et quel objet elle retourne. La fonction suppose que toutes vos images soient dans un répertoire appelé `data`, et donc utilisera le nom de fichier et créera le chemin complet (par exemple `data/ball.png`), en utilisant le module `os` pour s'assurer de la compatibilité entre plateforme différente (Linux, MacOS, Windows, ...). Ensuite elle essaye de charger l'image, et de convertir les régions alpha (ce qui vous permettra d'utiliser la transparence), et le cas échéant retourne une erreur *lisible par un être humain* si elle rencontre un problème. Finalement elle retourne un objet image, ainsi que son `rect`.

Vous pouvez créer des fonctions similaires pour le chargement de n'importe quelle autre ressource, tel que le chargement des sons. Vous pouvez aussi créer des classes de gestion de ressources, pour vous donner plus de flexibilité avec des ressources plus complexes. Par exemple, vous pouvez créer une classe `Music`, avec une fonction `__init__()` qui charge le son (peut-être en empruntant la fonction `load_sound()`), une méthode pour mettre en pause la musique, une méthode pour la redémarrer. Une autre classe de gestion de ressources utile peut être créée pour les connexions réseau. Des fonctions pour ouvrir des sockets, passer des données avec une sécurité appropriée et muni d'un contrôle d'erreur, fermer des sockets, finger (http://jargonf.org/wiki/finger) des adresses, ainsi que d'autres tâches concernant le réseau, pourront rendre l'écriture d'un jeu avec des capacités réseau moins pénible.

Souvenez-vous que la tâche première de ces fonctions/classes est de s'assurer qu'avec le temps, l'écriture des classes d'objet, et de la boucle principale, il n'y ait presque plus rien à faire. L'héritage de classes peut rendre ces classes de bases utiles. Ne vous emballez pas, des fonctions qui ne seront utilisées que par une classe devront être écrites dans cette classe, et non pas dans une fonction globale.

Classes d'objet de jeu

Une fois les modules chargées et les fonctions de gestion de ressources écrites, vous aimeriez écrire certains objets du jeu. La manière de le faire est très simple, bien qu'elle semble complexe au début. Vous devez écrire une classe pour chaque type d'objet du jeu, et ensuite vous pourrez créer une instance de ces classes pour chaque objet. Vous pourrez ensuite utiliser les méthodes de ces classes pour manipuler les objets, les déplacer et leur donner des capacités d'interaction. Votre jeu ressemblera donc à ceci (pseudo-code) :

```
#!/usr/bin/python
# coding: utf-8

[Charger vos modules ici]

[Fonctions de gestion des ressources ici]

class Ball:
    [fonctions de la balle (méthodes) ici]
    [par exemple, une fonction qui calcule une nouvelle
position]
    [et une fonction pour vérifier si elle touche les bords]
```

```
def main:
    [initier l environnement du jeu ici]

    [créer un nouvel objet, instance de la classe Ball]
    ball = Ball()

    while 1:
        [Vérifier les entrées utilisateur]

        [appel de la méthode update() de la balle]
        ball.update()
```

Ce n'est bien sûr qu'un exemple très simple, et vous aurez besoin d'y insérer tout le code nécessaire, en lieu et place des commentaires entre crochets. Mais vous devez connaître l'idée de base. Vous créez une classe, dans laquelle vous insérez toutes les fonctions d'une balle, en y incluant `__init__()`, qui créera tous les attributs d'une balle, et `update()`, qui déplacera la balle dans sa nouvelle position, avant de la transférer à l'écran dans cette position.

Vous avez la possibilité de créer d'autres classes pour tous les autres objets de jeu, et vous pourrez ensuite créer des instances pour chaque, et ainsi les gérer facilement à partir de la fonction `main` et/ou de la boucle du programme principal. Contrastez ceci avec le fait d'initialiser la balle dans la fonction `main`, et alors vous obtiendrez quantité de fonctions sans classes pour manipuler cet objet balle, et vous comprendrez heureusement pourquoi l'utilisation des classes est un avantage : cela vous permet de mettre tout le code de chaque objet à un seule endroit. Ceci rend l'utilisation des objets plus simple, et l'ajout de nouveaux objets et leur manipulation plus flexible. Au lieu d'ajouter plus de code pour chaque nouvel objet balle, vous pouvez simplement créer de nouvelles instances pour chaque nouvel objet balle. Magique !

Une simple classe balle

Voici une classe simple incluant les fonctions nécessaires pour la création d'un objet balle qui se déplacera sur l'écran si la fonction `update()` est appelée :

```
class Ball(pygame.sprite.Sprite):
    """Une balle qui se déplace sur lécran
    Retourne: objet ball
    Fonctions: update, calcNewPos
    Attributs: area, vector"""

    def __init__(self, vector):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('ball.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.vector = vector

    def update(self):
        newPos = self.calcNewPos(self.rect,self.vector)
        self.rect = newPos
```

```
    def calcNewPos(self,rect,vector):
        (angle,z) = vector
        (dx,dy) = (z*math.cos(angle),z*math.sin(angle))
        return rect.move(dx,dy)
```

Ici nous avons la classe `Ball`, avec une méthode `__init__()`, qui paramètre la balle, et une méthode `update()` qui change le rectangle de la balle pour une nouvelle position, et une méthode `calcNewPos()` pour calculer la nouvelle position de la balle basée sur sa position courante, et le vecteur par lequel elle se déplace. J'expliquerai la gestion de la physique dans un moment. La seule autre chose à noter est le *docstring*, qui est un peu plus long cette fois, et explique les bases de la classe. Ces chaînes de caractères sont utiles non seulement pour vous-même et les autres programmeurs qui lisent votre code, mais aussi pour les outils qui parsent votre code et le documentent. Elle ne feront pas la différence dans de petits programmes, mais dans les gros elles sont inestimables, c'est donc une bonne habitude à prendre.

Diversion 1 : Sprites

L'autre raison à la création d'une classe pour chaque objet est les sprites. Chaque image dont vous ferez le rendu dans votre jeu sera un objet sprite, et pour commencer : la classe de chaque objet devra hériter de la classe `Sprite`. L'héritage de classe est une fonctionnalité géniale de Python. À partir de maintenant la classe `Ball` possède toutes les méthodes de la classe `Sprite`, n'importe quelle instance d'objet de la classe `Sprite` sera enregistrée comme étant un sprite par Pygame. Tant que le texte et l'arrière-plan ne se déplacent pas, ça reste correct de transférer l'objet sur l'arrière-plan, Pygame manipule les objets sprite d'une manière différente que vous verrez lorsque nous examinerons le code du programme en entier.

En résumé, vous créez un objet `Ball` et un objet `Sprite` pour cette balle, et ensuite vous appelez la méthode `update()`sur l'objet `Sprite`, ce qui actualisera le sprite. Les sprites vous fournissent une manière sophistiquée de déterminer si deux objets sont en collision. Normalement vous pouvez simplement contrôler dans la boucle principale si leur rectangle se chevauchent, mais ca implique beaucoup de code qui sera inutile puisque la classe `Sprite` vous fournit spécialement les deux méthodes `spritecollide()` et `groupcollide()`.

Diversion 2 : Physique des vecteurs

Les autres choses à connaître à propos de ce code, en dehors de la structure de la classe `Ball`, ce sont les physiques de vecteur, utilisées pour calculer le mouvement de la balle. Dans n'importe quel jeu impliquant un mouvement angulaire, vous devez être à l'aise en trigonométrie, je ferais juste une petite introduction sur ce que vous devez savoir, pour comprendre la méthode `calcNewPos()`.

Pour commencer, vous avez remarqué que la balle possède un attribut `vector`, qui est construit à partir de `angle` et de `z`. L'angle est mesuré en radians et vous donne la direction dans laquelle se dirige la balle. `z` correspond à la vitesse à laquelle la balle se déplace. Ainsi en utilisant ce vecteur, nous pouvons déterminer la direction et la vitesse de la balle, et donc de combien elle doit se déplacer sur les axes X et Y.

Dans les bases mathématiques derrière les vecteurs, la partie gauche montre le mouvement projeté de la balle, représenté par la ligne bleue. La longueur de cette ligne (z) représente sa vitesse et l'angle est la direction dans elle se déplace. L'angle 0 pour le mouvement de la balle sera toujours pris dans le sens positif de l'axe des X (vers la droite), et sera mesuré dans le sens des aiguilles d'une montre comme vu sur le diagramme.

À partir de l'angle et de la vitesse de la balle, nous pouvons maintenant définir de combien s'est déplacée la balle le long des axes X et Y. Nous en avons besoin car Pygame n'inclut pas les calculs de vecteurs, et nous pouvons seulement déplacer la balle en bougeant son rectangle le long des deux axes. Nous avons donc besoin de faire correspondre l'angle et la vitesse en mouvement sur les axes X (dx) et Y (dy). C'est une simple question de géométrie, et peut être obtenue grâce aux formules du diagramme.

Si vous avez étudié la trigonométrie élémentaire auparavant, rien ne sera nouveau pour vous. Mais au cas où vous l'auriez oubliée, voici quelques formules indispensables qui vous aideront à visualiser les angles (je trouve plus facile de visualiser les angles en degrés plutôt qu'en radians).

$$\theta_{deg} = \theta_{rad} \cdot \frac{180}{\pi} \qquad et \qquad \theta_{rad} = \theta_{deg} \cdot \frac{\pi}{180}$$

Objets contrôlés par l'utilisateur

Pour l'instant, vous avez créé une fenêtre Pygame, et fait un rendu d'une balle qui se déplace sur l'écran. La prochaine étape est de créer quelques raquettes qui soient sous le contrôle de l'utilisateur. C'est potentiellement plus simple que la balle, car çà ne requiert aucune physique. Toutefois ceci ne se vérifie plus lorsque votre objet possède des déplacements plus complexe que haut et bas, par exemple dans un jeu de plateforme comme Mario, auquel cas vous aurez besoin de mettre en jeu de la physique. Les objets contrôlables sont simples à mettre en œuvre, remerciez Pygame pour son système de file d'évènements, comme vous pourrez le voir.

Une simple classe Bat

Le principe derrière la classe Bat est similaire à la classe Ball. Vous avez besoin d'une méthode `__init__()` pour initialiser la raquette (vous pourrez donc créer des instances d'objet pour chaque raquette), d'une méthode `update()` pour appliquer les changements sur la raquette avant de la transférer à l'écran, et diverses autres méthodes qui définiront ce que fait cette classe. Voici un échantillon du code :

```
class Bat(pygame.sprite.Sprite):
    """Raquette de 'tennis' déplaçable qui peut frapper la balle
    Retourne: objet bat
    Méthode: reinit, update, moveup, movedown
    Attributs: which, speed"""

    def __init__(self, side):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('bat.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.side = side
        self.speed = 10
        self.state = "still"
        self.reinit()

    def reinit(self):
        self.state = "still"
        self.movepos = [0,0]
```

```
        if self.side == "left":
            self.rect.midleft = self.area.midleft
        elif self.side == "right":
            self.rect.midright = self.area.midright

    def update(self):
        newpos = self.rect.move(self.movepos)
        if self.area.contains(newpos):
            self.rect = newpos
        pygame.event.pump()

    def moveup(self):
        self.movepos[1] = self.movepos[1] - (self.speed)
        self.state = "moveup"

    def movedown(self):
        self.movepos[1] = self.movepos[1] + (self.speed)
        self.state = "movedown"
```

Comme vous pouvez le voir, cette classe est très similaire à la classe Ball dans sa structure. Mais les différences se situent dans ce que fait chaque fonction. Tout d'abord, il y a une méthode `reinit()` qui est utilisée lorsqu'un round est terminé : la raquette retourne dans sa position de départ, et chacun de ses attributs à ses valeurs d'origine. Ensuite, la manière dont la raquette bouge est un peu plus complexe que la balle, étant donné que ses mouvements sont simples (haut/bas) mais dépendent de ce que désire l'utilisateur, tandis que la balle conservera son mouvement à chaque image. Pour mettre en évidence la façon dont la raquette bouge, il est pratique d'examiner ce petit diagramme pour voir la séquence des évènements :

Le joueur enfonce la touche ↑ ⇒ self.state = "moving" self.moveup() ⇒ Le joueur relache la touche ⇒ self.state = "still" self.movepos = [0,0]

C'est ce qui se passe ici si la personne qui contrôle la raquette enfonce la touche qui fait se déplacer la raquette vers le haut. À chaque itération de la boucle principale du jeu (à chaque image), si la touche est maintenue enfoncée, alors l'attribut `state` de cet objet raquette sera paramétré à "moving", et la méthode `moveup()` sera appelée, causant la réduction de la position Y de la raquette d'une valeur correspondant à l'attribut `speed` (dans cet exemple 10). En d'autre mots, tant que la touche reste enfoncée, la raquette se déplacera à l'écran de 10 pixels par image. L'attribut `state` n'est pas utilisé ici, mais c'est très utile de le connaître si vous désirez appliquer des effets à la balle, ou si vous utilisez une sortie pour le débogage.

Diversion 3 : évènements Pygame

Alors, comment allons nous savoir quand le joueur est en train d'enfoncer la touche, ou la relâche ? Avec le système de file d'évènement de Pygame, pardi! C'est un système vraiment simple à utiliser et à comprendre, çà ne devrait pas être long :) Vous avez déjà observé la file d'évènement en action dans le programme Pygame de base, où elle était utilisée pour vérifier si l'utilisateur voulait quitter l'application. Le code pour déplacer la raquette est aussi simple que çà :

```
for event in pygame.event.get():
    if event.type == QUIT:
```

```
            return
        elif event.type == KEYDOWN:
            if event.key == K_UP:
                player.moveup()
            if event.key == K_DOWN:
                player.movedown()
        elif event.type == KEYUP:
            if event.key == K_UP or event.key == K_DOWN:
                player.movepos = [0,0]
                player.state = "still"
```

Ici nous supposons que vous avez déjà créé une instance de `Bat`, et appelé l'objet `player`. Vous pouvez observer la couche familière de la structure `for`, qui produit une itération à chaque évènement trouvé dans la file d'évènement de Pygame, eux-même retrouvés grâce à la fonction `event.get()`. L'utilisateur enfonce une touche, appuie sur le bouton de la souris, ou bouge le joystick, toutes ces actions seront placées dans la file d'évènement de Pygame, et conservées jusqu'à leur utilisation. Donc à chaque itération de la boucle de jeu principale, vous irez faire un tour dans ces évènements vérifier s'il y en a quelques uns que vous pouvez utiliser. La fonction `event.pump()` qui était dans la méthode `Bat.update()` est appelée à chaque itération pour *pomper* les vieux évènements et garder la file à jour.

D'abord nous vérifions si l'utilisateur veut quitter le programme, et si oui on quitte le programme. Ensuite nous vérifions si une touche est enfoncée, et si oui, nous vérifions si elle correspond à une des touches affectée au déplacement de la raquette, si oui alors nous appelons la méthode de déplacement appropriée, et définissons l'état du joueur. À travers les états "moveup" et "movedown", modifiés par les méthodes `moveup()` et `movedown()`, nous produisons un code plus soigné et nous ne cassons pas l'*encapsulation*, ce qui signifie que vous assignez des attributs aux objets eux-mêmes, sans se référer au nom de l'instance de cet objet. Remarquez que nous avons 3 états : "still", "moveup" et "mouvedown". Encore une fois, ils deviennent utiles si vous voulez débugguer ou calculer un effet sur la balle. Nous vérifions aussi si une touche est "partie" (si elle n'est plus enfoncée), et alors si c'est la bonne touche, nous stoppons le déplacement de la raquette.

Assembler le tout

Bien, vous avez appris toutes les bases nécessaires pour écrire un petit jeu. Vous devez devriez avoir compris comment créer un objet Pygame, comment Pygame affiche les objets, comment manipuler les évènements, et comment vous pouvez utiliser la physique pour introduire des animations dans votre jeu. Maintenant je vais vous montrer comment vous pouvez prendre toutes ces morceaux de code et les assembler dans un jeu qui fonctionne. Ce que nous avons besoin tout d'abord, c'est de faire rebondir la balle sur les bords de l'écran, et que la raquette aussi soit capable de faire rebondir la balle, en d'autres termes, ce ne sera pas un jeu très compliqué. Pour ce faire, nous utiliserons les méthodes de collision de Pygame.

Faire rebondir la balle sur les bords de l'écran

La principe de base de ce type de rebond est simple à comprendre. Vous prenez les coordonnées des 4 coins de la balle, et vous vérifiez s'ils correspondent avec les coordonnées X et Y des bords de l'écran. Donc si les coins haut-gauche et haut-droit ont leur coordonnée Y à 0, vous savez que la balle est actuellement contre le bord haut de l'écran. Nous ferons tout cela dans la fonction `update()`, après que nous ayons défini la nouvelle position de la balle.

```
if not self.area.contains(newPos):
    tl = not self.area.collidepoint(newPos.topleft)
    tr = not self.area.collidepoint(newPos.topright)
    bl = not self.area.collidepoint(newPos.bottomleft)
    br = not self.area.collidepoint(newPos.bottomright)
    if tr and tl or (br and bl):
        angle = -angle
    if tl and bl:
        self.offcourt(player=2)
    if tr and br:
        self.offcourt(player=1)

self.vector = (angle,z)
```

Ici nous contrôlons que la variable `area` contient la nouvelle position de la balle. Elle devrait toujours être vraie, nous n'aurons donc pas besoin de la clause `else`, bien que dans d'autres circonstances vous devriez considérer ce cas de figure. Nous contrôlons alors si les coordonnées des quatre coins entrent en collision avec les bords de l'écran, et créons des objets pour chaque résultat. Si c'est vérifié, les objets auront leur valeur à 1, ou `true`. Sinon, la valeur sera `None`, ou `false`. Nous verrons alors si elle touche le dessus ou le dessous, et si oui nous changerons la direction de la balle. En pratique, grâce à l'utilisation des radians, nous pourrons le faire facilement, juste en inversant sa valeur (positif/négatif). Nous contrôlons aussi que la balle ne traverse pas les bords, ou alors nous appellerons la fonction `offcourt()`. Ceci dans mon jeu, replace la balle, ajoute 1 point au score du joueur spécifié lors de l'appel de la fonction, et affiche le nouveau score.

Enfin, nous réaffectons le vecteur basé sur le nouvel angle. Et voilà, la balle rebondiras gaiement sur les murs et sortira du court avec un peu de chance.

Faire rebondir la balle sur la raquette

Faire en sorte que la balle rebondisse sur les raquettes est similaire au rebonds sur les bords de l'écran. Nous utilisons encore la méthode des collisions, mais cette fois, nous vérifions que les rectangles de la balle entrent en collision avec ceux des raquettes. Dans ce code, nous y ajoutons des suppléments pour éviter certains problèmes. Vous devriez trouver tout un tas de codes supplémentaires à ajouter pour éviter certains problèmes ou bugs, il est plus simple de les trouver lors de l'utilisation.

```
else:
    # Réduire les rectangles pour ne pas frapper la balle
derrière les raquettes
    player1.rect.inflate(-3, -3)
    player2.rect.inflate(-3, -3)

    # Est-ce que la raquette et la balle entre en collision ?
    # Notez que je mets dans une règle à part qui définit
self.hit à 1 quand ils entrent en collision
    # et à 0 à la prochaine itération. C'est pour stopper un
comportement étrange de la balle
```

```
    # lorsqu'il trouve une collision *à l'intérieur* de la
raquette, la balle s'inverse, et est
    # toujours à l'intérieur de la raquette et donc rebondit à
l'intérieur.
    # De cette façon, la balle peut toujours s'échapper et
rebondir correctement
    if self.rect.colliderect(player1.rect) == 1 and not
self.hit:
        angle = math.pi - angle
        self.hit = not self.hit
    elif self.rect.colliderect(player2.rect) == 1 and not
self.hit:
        angle = math.pi - angle
        self.hit = not self.hit
    elif self.hit:
        self.hit = not self.hit
self.vector = (angle,z)
```

Nous débutons cette section à partir de la condition `else`, à cause de la partie précédente du code qui vérifie si la balle frappe les bords. Cela prend tout son sens si elle ne frappe pas les bords, elle pourrait frapper une raquette, donc nous poursuivons la condition précédente. Le premier problème est de réduire le rectangle des joueurs de 3 pixels dans les deux dimensions pour empêcher la raquette de frapper la balle lorsqu'elle est derrière elle. Imaginez que vous venez juste de bouger la raquette et que la balle se trouvait derrière elle, les rectangles se chevauchent, et la balle sera considérée comme "frappée", c'est pour prévenir ce petit bug.

Ensuite nous vérifions si les rectangles entrent en collision, avec une correction de bug de plus. Remarquez que j'ai commenté toute cette partie de code, c'est toujours préférable d'expliquer une partie du code qui parait obscure, que ce soit pour les autres qui regardent votre code, ou pour vous lorsque vous y reviendrez plus tard. Sans la correction du bug, la balle pourra heurter un coin de la raquette, changer de direction, et l'image d'après, trouver qu'elle est toujours à l'intérieur de la raquette. Alors le programme pensera que la balle est frappée une nouvelle fois et rechangera de direction. Cela peut survenir plusieurs fois, ce qui rendrait le comportement de la balle complètement irréaliste. Donc nous avons une variable, `self.hit` qui sera définie à `true` quand elle sera frappée, et à `false` une image plus tard. Quand nous vérifions si les rectangles sont entrés en collision, nous contrôlons si `self.hit` est à `true` ou `false` pour stopper les rebonds internes.

L'imposant code ici est facile à comprendre. Tous les rectangle possèdent une fonction `colliderect()`, dans laquelle vous envoyez le rectangle d'un autre objet, qui retournera `1` (`true`) si les rectangles se chevauchent, et `0` (`false`) dans le cas contraire. En cas de réponse positive, nous pouvons changer la direction en soustrayant l'angle actuel par π (encore un petit truc que vous pouvez faire avec les radians, qui ajustera l'angle de 90 degrés et renverra la bonne direction. Vous pourrez trouver qu'à ce stade la compréhension approfondie des radians soit un exigeance!). Pour terminer le contrôle du bug, nous repassons `self.hit` à `false` à partir de l'image qui suit la frappe de la balle.

Nous réaffectons alors le vecteur. Vous aimeriez bien sur enlever la même ligne dans la précédente partie de code, mais vous ne pourrez le faire qu'après le test conditionnel `if-else`. Et ce sera tout. Le code assemblé permettra maintenant à la balle de frapper les côtés et les raquettes.

Le produit final

Le produit final, avec toutes les parties de code assemblées grâce à du code glu ressemblera à ceci :

> Code final de Tom's Pong

Comme vous avez pu voir le produit fini, je vais vous rediriger vers *Tom's Pong*, jeu sur lequel est basé ce tutoriel. Téléchargez-le, étudiez le code source, et vous verrez une implémentation de Pong utilisant tout le code que vous avez vu dans ce tutoriel, ainsi que d'autres ajouts fait dans des versions précédentes, comme des propriétés physique pour les effets et ainsi que d'autres résolutions de bugs.

Modes d'affichage

Paramétrer les modes d'affichage

Traduit de l'anglais, document original par "Pete Shinners" :

https://www.pygame.org/docs/tut/DisplayModes.html

Introduction

La définition du mode d’affichage dans pygame crée une surface d’image visible sur le moniteur. Cette surface peut couvrir tout l'écran ou être fenêtrée sur des plates-formes prenant en charge un gestionnaire de fenêtres. La surface d’affichage n’est rien d’autre qu’un objet surfacique pygame standard . `pygame.display`. Le module pygame contient des fonctions spéciales permettant de contrôler la fenêtre d’affichage et le module d’ écran pour maintenir le contenu de la surface de l’image à jour sur le moniteur.

Définir le mode d'affichage dans pygame est une tâche plus facile qu'avec la plupart des bibliothèques graphiques. L'avantage est que si votre mode d'affichage n'est pas disponible, pygame émulera le mode d'affichage que vous avez demandé. Pygame sélectionnera une résolution d'affichage et une profondeur de couleur qui correspondent le mieux aux paramètres que vous avez demandés, puis vous permettra d'accéder à l'écran avec le format que vous avez demandé. En réalité, le pygame.displaymodule pygame permettant de contrôler la fenêtre d’affichage et le module d’ écran constituant une liaison entre la bibliothèque SDL et SDL, SDL effectue tout ce travail.

Il y a des avantages et des inconvénients à régler le mode d'affichage de cette manière. L'avantage est que si votre jeu nécessite un mode d'affichage spécifique, il s'exécutera sur des plates-formes ne prenant pas en charge vos besoins. Cela facilite également la vie lorsque vous commencez quelque chose, il est toujours facile de revenir en arrière et de rendre le choix du mode un peu plus précis. L'inconvénient est que ce que vous demandez n'est pas toujours ce que vous obtiendrez. Il y a aussi une pénalité de performance lorsque le mode d'affichage doit être émulé. Ce didacticiel vous aidera à comprendre les différentes méthodes d'interrogation des capacités d'affichage de la plate-forme et à définir le mode d'affichage de votre jeu.

Définir les bases

La première chose à apprendre est de savoir comment définir le mode d'affichage actuel. Le mode d'affichage peut être défini à tout moment après que le `pygame.display` pour contrôler la fenêtre d'affichage et le module d' écran ont été initialisés. Si vous avez précédemment défini le mode d’affichage, le ré-activer modifiera le mode actuel. Le réglage du mode d'affichage est géré avec la fonction Initialiser une fenêtre ou un écran pour l'affichage . Le seul argument requis dans cette fonction est une séquence contenant la largeur et la hauteur du nouveau mode d'affichage. L'indicateur de profondeur correspond aux bits demandés par pixel pour la surface. Si la profondeur donnée est 8, pygame créera une surface mappée en couleur. Lorsque la profondeur de bit est plus élevée, `pygamepygame.display.set_mode((width, height), flags, depth)` utilisera un mode couleur compacté. Vous trouverez beaucoup plus d'informations sur les profondeurs et les modes de couleur dans la documentation des modules d'affichage et de surface. La valeur par défaut pour la profondeur est 0. Lorsque l'argument 0 est attribué , pygame sélectionne la meilleure résolution à utiliser, généralement identique à la profondeur actuelle du système. L'argument flags vous permet de contrôler des

fonctionnalités supplémentaires pour le mode d'affichage. Vous pouvez créer la surface d'affichage dans la mémoire matérielle avec le flag `HWSURFACE`. Encore une fois, vous trouverez plus d'informations à ce sujet dans les documents de référence pygame .

Comment décider

Alors, comment choisir un mode d'affichage qui fonctionnera le mieux avec vos ressources graphiques et la plate-forme sur laquelle tourne votre jeu? Il existe plusieurs méthodes pour collecter des informations sur le périphérique d'affichage. Toutes ces méthodes doivent être appelées après l'initialisation du module d'affichage, mais vous souhaiterez probablement les appeler avant de définir le mode d'affichage. Tout d'abord, `pygame.display.Info()` renverra un type d'objet spécial de **VidInfo**, qui peut vous en apprendre beaucoup sur les capacités du pilote graphique. La fonction Obtenir la liste des modes plein écran disponibles peut être utilisée pour rechercher les modes graphiques pris en charge par le système. Choisir la meilleure profondeur de couleur pour un mode d'affichage prend les mêmes arguments que `pygame.display.set_mode()`, mais renvoie la résolution en bits la plus proche de celle demandée. Enfin, `pygame.display.get_driver()` renvoie le nom du pilote graphique sélectionné par pygame .

Rappelez-vous juste la règle d'or. Pygame fonctionnera avec pratiquement tous les modes d'affichage demandés. Certains modes d'affichage devront être émulés, ce qui ralentira votre jeu, car pygame devra convertir chaque mise à jour effectuée en mode "réel". Le mieux est de toujours laisser pygame choisir la meilleure résolution en bits et de convertir toutes vos ressources graphiques dans ce format lors de leur chargement. Vous pouvez laisser pygame choisir sa résolution en appelant set_mode() sans argument de profondeur ou avec une profondeur de 0, ou vous pouvez appeler mode_ok() pour trouver la résolution en bits la plus proche de ce dont vous avez besoin.

Lorsque votre mode d'affichage est fenêtré, vous devez généralement calculer la même profondeur de bits que le bureau. Lorsque vous êtes en plein écran, certaines plates-formes peuvent basculer vers la résolution la plus adaptée à vos besoins. Vous pouvez trouver la profondeur du bureau actuel si vous obtenez un objet VidInfo avant de définir votre mode d'affichage.

Après avoir défini le mode d'affichage, vous pouvez obtenir des informations sur ses paramètres en obtenant un objet VidInfo ou en appelant l'une des méthodes Surface.get * sur la surface d'affichage.

Fonctions

Ce sont les routines que vous pouvez utiliser pour déterminer le mode d'affichage le plus approprié. Vous trouverez plus d'informations sur ces fonctions dans la documentation du module d'affichage.

pygame.display.mode_ok(size, flags, depth)

Cette fonction prend exactement les mêmes arguments que pygame.display.set_mode (). Il renvoie la meilleure résolution disponible pour le mode que vous avez décrit. Si cela renvoie zéro, le mode d'affichage souhaité n'est pas disponible sans émulation.

pygame.display.list_modes(depth, flags)

Obtenir la liste des modes plein écran disponibles

Renvoie une liste des modes d'affichage pris en charge avec la profondeur et les indicateurs demandés. Une liste vide est renvoyée lorsqu'il n'y a pas de mode. L'argument flags est défini par défaut sur FULLSCREEN. Si vous spécifiez vos propres indicateurs sans FULLSCREEN, vous obtiendrez probablement une valeur de retour de -1. Cela signifie que toute taille d'affichage convient, car l'affichage sera fenêtré. Notez que les modes listés sont triés du plus grand au plus petit. pygame.display.Info()Créer un objet d'information d'affichage vidéo

Cette fonction retourne un objet avec de nombreux membres décrivant le périphérique d'affichage. L'impression de l'objet VidInfo vous montrera rapidement tous les membres et toutes les valeurs pour cet objet.

```
>>> import pygame.display
>>> pygame.display.init()
>>> info = pygame.display.Info()
>>> print info
<VideoInfo(hw = 1, wm = 1,video_mem = 27354
           blit_hw = 1, blit_hw_CC = 1, blit_hw_A = 0,
           blit_sw = 1, blit_sw_CC = 1, blit_sw_A = 0,
           bitsize  = 32, bytesize = 4,
           masks =  (16711680, 65280, 255, 0),
           shifts = (16, 8, 0, 0),
           losses =  (0, 0, 0, 8)>
```

Vous pouvez tester tous ces indicateurs en tant que membres de l'objet VidInfo. Les différents indicateurs de fusion indiquent si l'accélération matérielle est prise en charge lors de la fusion des différents types de surfaces sur une surface matérielle.

Exemples

Voici quelques exemples de différentes méthodes pour initialiser l’affichage graphique. Ils devraient vous aider à avoir une idée de la façon de régler votre mode d’affichage.

```
>>> #give me the best depth with a 640 x 480 windowed display
>>> pygame.display.set_mode((640, 480))

>>> #give me the biggest 16-bit display available
>>> modes = pygame.display.list_modes(16)
>>> if not modes:
...     print '16-bit not supported'
... else:
...     print 'Found Resolution:', modes[0]
...     pygame.display.set_mode(modes[0], FULLSCREEN, 16)

>>> #need an 8-bit surface, nothing else will do
>>> if pygame.display.mode_ok((800, 600), 0, 8) != 8:
...     print 'Can only work with an 8-bit display, sorry'
```

```
... else:
...     pygame.display.set_mode((800, 600), 0, 8)
```

Documentation de référence

Pygame : le module de haut niveau contenant le reste des fonctions

- Introduction à Pygame
- Importation et initialisation
- Introduction au parcours de tableau
- Déplacer une image
- Chimp - Ligne par ligne
- Introduction au module Sprite
- Introduction au module Surfarray
- Guide du débutant
- Concevoir des jeux avec Pygame
    - Code final de Tom's Pong

Le package de haut niveau de Pygame.

```
pygame.init - initialise tout les
modules de Pygame importés
pygame.quit - quitte
convenablement tous les modules
de Pygame importés
pygame.error - Exception Pygame
standard
pygame.get_error - Récupère le
message de l'erreur actuelle
pygame.get_sdl_version - récupère
le numéro de version de la
bibliothèque SDL
pygame.get_sdl_byteorder -
récupère l'ordre des octets dans
la SDL
pygame.register_quit - Contient
la fonction à appeler lorsque
Pygame quitte
pygame.version - petit module
contenant quelques informations
de version
```

Le paquet Pygame représente le paquet de haut niveau qui contient les autres. Pygame lui-même, est décomposé en beaucoup de sous-modules, mais cela ne doit pas affecter les programmes qui utilisent Pygame.

Pour des raisons pratiques, la plupart des variables internes à pygame ont été placées à l'intérieur d'un module nommé 'pygame.locals'. Cela signifie qu'il doit être utilisé en écrivant 'from pygame.locales import *', en supplément de 'import pygame'.

Avec l'instruction 'import pygame', tous les sous modules de Pygame disponibles sont automatiquement importés. Soyez conscient que quelques uns des modules de Pygame sont considérés comme "optionnels", et peuvent ne pas être disponibles. Dans ce cas, Pygame fournira un objet pour le remplacer, qui peut être utilisé pour tester la disponibilité.

pygame.init

pygame.init

initialise tous les modules de Pygame déjà importés

```
pygame.init(): return (numpass, numfail)
```

Initialise tous les modules de Pygame déjà importés. Aucune exception ne sera levée si un module échoue, mais le nombre total de succès et d'échecs sera renvoyé dans un tuple. Vous pouvez toujours initialiser les modules individuels manuellement, mais pygame.init est un moyen approprié pour tout faire démarrer. La fonction init() pour les modules individuels lèvera une exception si elle échoue.

Vous pourriez vouloir initialiser les différents modules séparément pour accélérer votre programme, ou ne pas utiliser certaines choses dont votre jeu n'a pas besoin. Il est sans danger d'appeler cette méthode init() plus d'une fois. Des appels répétés n'auront aucun effet. Ceci est vrai même si vous avez déjà appelé pygame.quit() (déinitialise tous les modules pygame).
