université
PARIS-SACLAY

FACULTÉ
DES SCIENCES
D'ORSAY

PROJET PCII

Réalisation d'un jeu vidéo de type « course de voiture ».

Janethe Ferhoune – Hawa Ina Ouro Diallo

TABLE DES MATIERES

Contenu

Introduction

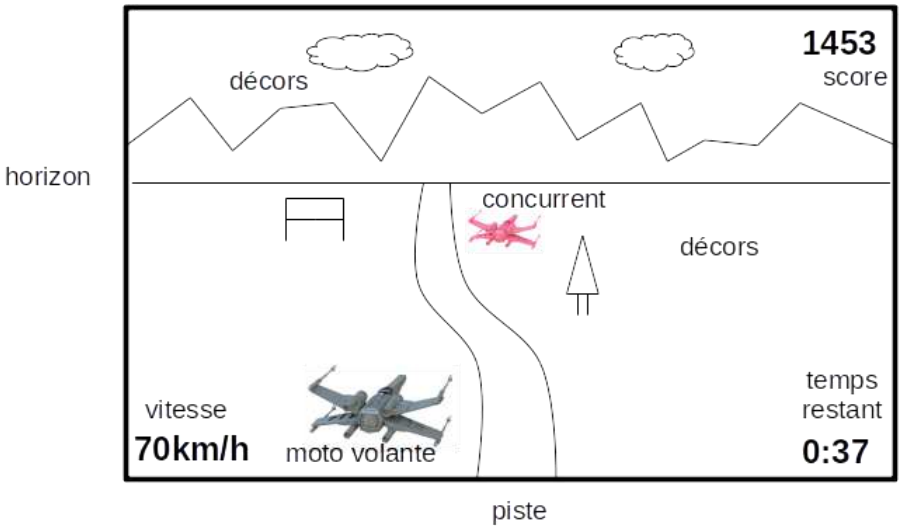
L'objectif de notre projet est de réaliser un jeu vidéo de type « course de voiture » inspiré des années 80. Ce jeu disposera d'une particularité : le véhicule ne pourra se déplacer qu'horizontalement à l'aide des touches directionnelles. Sa vitesse est gérée automatiquement à partir de sa proximité par rapport à la piste, ainsi, plus le véhicule est éloigné de la piste et plus celui-ci perdra progressivement de la vitesse jusqu'à finalement atteindre zéro et mettre fin à la course. Dans le cas contraire, s'il est positionné sur la piste, sa vitesse augmentera.

Pour compliquer la chose, le joueur dispose d'un temps imparti pour atteindre des points de contrôle qui lui permettront de s'ajouter du temps. Si le temps imparti atteint zéro, la partie se termine aussitôt.

Au cours de sa course, le joueur peut croiser des concurrents et a la possibilité de les dépasser, ce qui aura pour conséquence d'augmenter son score.

Le but du jeu consiste donc à avoir le score le plus élevé et pour cela, le joueur doit faire en sorte de rester le plus possible en course.

Voici une représentation du but à atteindre une fois le jeu finalisé :



Au cours de la recherche de sprites pour la représentation du personnage, des décors et des obstacles, nous sommes tombées sur des images appartenant à l'univers de « *Naruto Shippuden* », c'est pourquoi nous avons opté, pour la représentation du véhicule, de choisir le personnage de « *Naruto* ».

Analyse globale

Ce projet englobe un ensemble de fonctionnalités assez diverses allant de l'implémentation de la vue à la mise en place d'obstacles, d'adversaires, d'un processus de calcul de la vitesse et de l'accélération selon la position du véhicule par rapport à la piste. Les différentes fonctionnalités varient selon leur spécificité et leur difficulté d'implémentation.

Nous avons listé ci-dessous les différentes sous-fonctionnalités à implémenter au sein de ce projet :

- ✓ Un horizon surmonté d'un décor ;
- ✓ Une piste infinie limitée par l'horizon et générée aléatoirement ;
- ✓ Un mécanisme de contrôle au clavier pour déplacer le véhicule horizontalement ;
- ✓ Un temps restant et un kilométrage ;
- ✓ Un ajout de temps, qui diminue progressivement, à chaque point de contrôle atteint ;
- ✓ Une fin de partie dans le cas où la vitesse du personnage ou le temps restant atteint zéro ;
- ✓ Un ajout de décors autour de la piste ;
- ✓ Un ajout d'un écran d'accueil ;
- ✓ Une possibilité de contrôler la vitesse du véhicule.

Plan de développement

Conception générale

Pour le développement de notre interface graphique, nous avons adopté un modèle MVC (Modèle, Vue, Contrôleur). En partant de ce principe, qui consiste à séparer le code de notre interface graphique en trois parties distinctes dans le but de structurer celui-ci, nous avons ajouté, au sein de notre projet, trois packages représentant chacun une des parties du modèle MVC : « *model* », « *view* » et « *control* ». Nous avons ensuite regroupé respectivement au sein de ceux-ci :

- ⇨ Les classes qui regroupent l'ensemble des données qui caractérisent l'état de notre interface ;
- ⇨ Les classes qui représentent le rendu de l'état du modèle à l'utilisateur ;
- ⇨ Les classes qui définissent la manière dont l'état du modèle change.

Voici les différentes classes présentent au sein de notre interface graphique :

ControlAccueil	Provoque l'ouverture de la fenêtre de jeu, l'affichage des scores ou la fermeture de l'application lorsque le bouton correspondant est cliqué.
ControlFenetrePrincipale	Provoque la fermeture de la fenêtre de jeu et rend visible la fenêtre d'accueil.
ControlPersonnage	Provoque la modification de la valeur en abscisse du personnage lorsque la touche directionnelle gauche ou droite est appuyée/enfoncée.
ControlSouris	Provoque l'affichage d'une image au sein d'un bouton lorsque la souris survole le composant associé à ce contrôleur.
AudioJeu	Gère le son.
Avancer	Incrémente la valeur de décalage afin de simuler l'avancement du personnage et incrémente le score du joueur.
Choji	Classe représentative d'un personnage immobile (qui ne se déplace pas) et qui représente un décor positionné au niveau de l'horizon.
Decompte	Modifie le temps restant au joueur pour atteindre le prochain point de contrôle.
Decor	Interface contenant des méthodes appliquées aux décors.
DecorImmobile	Classe représentative d'un décor qui n'a aucune incidence sur le personnage du joueur. Récupère les images associées au décor, modifie l'état du décor et initialise les positions en abscisse et ordonnée du décor.
DecorMobile	Classe représentative d'un décor qui n'a aucune incidence sur le personnage du joueur . Récupère les images associées au décor, modifie l'état du décor, initialise les positions en abscisse et ordonnée du décor et déterminer le côté vers lequel le décor se déplace.
DeidaraTobi	Classe représentative d'un lot de personnages qui se déplace et qui représente un décor qui n'a aucune incidence sur le personnage.
EtatAccueil	Détermine les actions effectuées aux boutons de la fenêtre d'accueil et permet le lancement de la fenêtre de jeu, l'affichage des scores et l'arrêt de l'application.
EtatDecorHorizon	Genère ine liste de décors positionnés au niveau de l'horizon.
EtatNaruto	Modifie l'état et la position du personnage.
EtatObstacke	Genère et met ç jour une liste d'obstacles.
EtatPointControle	Génère un point de contrôle lorsque le score du joueur atteint un certain score.
EtatProfilNaruto	Modifie l'état du profil du personnage lorsque le personnage atteint le point de contrôle ou est touché par un obstacle.
FinPartie	Détermine la fin de partie lorsque la vitesse du personnage ou le temps restant atteint zéro et affiche le score final du joueur.
Gai	Classe représentative d'un personnage immobile et qui représente un décor.
HighScore	Mémorise les dix meilleurs scores du joueur.

Obstacle	Récupère des images représentatives de l'obstacle, détermine lorsque le personnage est touché par un obstacle et initialise les positions en ordonnée et abscisse de l'obstacle
Piste	Genère une liste de points permettant le traçage de la piste et détermine si le perosnnage est positionné sur le piste et incrémente la vitesse du personnage en conséquence.
PointControle	Récupère les images représentatives du point de contrôle, détermine lorsque le personnage a atteint le point de contrôle et initialise les positions en ordonnée et abscisse du point de contrôle.
Score	Gère le score du joueur en l'incrémentant ou en le décrémentant.
Vitesse	Fère la vitesse du personnage en l'incrémentent ou en le décrémentant.
Affichage	Composant sur lequel est dessiné un horizon, des décors, une piste, un personnage, le profil du personnage, sa vitesse, le score du joueur et le temps restant au joueur pour atteindre le prochain point de contrôle.
FenetreAccueil	Feneêtre sur laquelle est affichée un composant contenant des boutons permettant d'accéder à la suite de l'application.
FenetrePrincipale	Fenêtre principale de l'application dans laquelle est affichée le composant permettant d'afficher le jeu.
VueAccueil	Composant contenant un fond d'écran et des boutons permettant d'accéder à la suite de l'application.
VueDecor	Dessine le fond d'écran de l'horzion, la pelouse et les décors de l'horizon au sein du composant.
VueNaruto	Dessine le personnage au sein du composant.
VueObstacle	Dessine des obstacles au sein du composant.
VuePiste	Dessine la piste au sein du composant.
VuePointControle	Dessine le point de contrôle au sein du composant.
VueProfilNaruto	Dessine le profil du personnage au sein du composant.

Et voici la manière dont nous les avons séparé afin d'être conforme au modèle MVC :

	Modèle	Vue	Contrôleur
ControlAccueil			X
ControlFenetrePrincipale			X
ControlPersonnage			X
ControlSouris			X
AudioJeu	X		
Avancer	X		
Choji	X		
Decompte	X		
Decor	X		
DecorImmobile	X		
DecorMobile	X		
DeidaraTobi	X		
EtatAccueil	X		
EtatDecorHorizon	X		
EtatNaruto	X		
EtatObstacke	X		
EtatPointControle	X		
EtatProfilNaruto	X		
FinPartie	X		
Gai	X		
HighScore	X		
Obstacle	X		
Piste	X		
PointControle	X		
Score	X		
Vitesse	X		
Affichage		X	
FenetreAccueil		X	
FenetrePrincipale		X	
VueAccueil		X	
VueDecor		X	
VueNaruto		X	
VueObstacle		X	
VuePiste		X	
VuePointControle		X	
VueProfilNaruto		X	

Conception détaillée

Tout d'abord, l'ensemble des images utilisées au sein de notre interface graphique se trouvent dans le dossier « *ressource* », qui se trouve à la racine de notre projet.

DESSIN DE L'HORIZON AU SEIN D'UNE FENETRE

Nous avions, au début du projet, représenté l'horizon par une ligne horizontale. Par la suite, nous avons remplacé cette ligne par une image fixe délimitant la piste du champ de décors. Nous avons donc recherché une image qui puisse correspondre et avons opté pour l'image ci-dessous :

La création de la fenêtre a nécessité la création de la classe « *FenetrePrincipale* » qui hérite de la classe « *JFrame* ». Afin de représenter le composant contenant le dessin de l'horizon, nous avons également ajouté à notre projet la classe « *Affichage* » qui hérite de la classe « *JPanel* ». Nous l'avons ensuite ajouté au sein de la fenêtre à l'aide de la méthode « *add* » de la classe « *JFrame* ».

Afin d'afficher l'image de l'horizon, nous avons créé une nouvelle classe nommée « *VueDecor* ». Dans celle-ci, nous commençons tout d'abord par récupérer l'image ci-dessus au sein de l'attribut « *fond* ». Nous avons ensuite créé une méthode nommée « *drawBackground* » dans laquelle nous faisons appel à la méthode « *drawImage* » de la classe « *Graphics* » qui prend cette image en argument afin de l'afficher.

Cette méthode est ensuite appelée au sein de la classe « *Affichage* », dans la méthode réécrite « *paint* » afin d'afficher l'image au sein du composant principal de notre interface.

ELEMENTS DE DECORS A L'HORIZON

Les sprites suivants représentent les décors et appartiennent tous à l'univers de « *Naruto Shippuden* » :




Comme les décors possèdent tous des méthodes similaires, nous avons créé l'interface « *Decor* » qui possèdent les méthodes suivantes :

- ⇨ *« getPositionX »*, qui renvoie la position en abscisse du décor ;
- ⇨ *« getPositionY »*, qui renvoie la position en ordonnée du décor ;
- ⇨ *« getImage »*, qui renvoie l'image du décor ;
- ⇨ *« getWidth »*, qui renvoie la largeur de l'image ;
- ⇨ *« getHeight »*, qui renvoie la hauteur de l'image ;
- ⇨ *« setEtat »*, qui modifie l'image affichée ;
- ⇨ *« loadImages »*, qui récupère les différentes images du décor.

Nous devions ensuite définir deux nouvelles classes afin de représenter un décor immobile et un décor immobile. Nous avons donc créé la classe « *DecorImmobile* » qui hérite de la classe « *Thread* » et implémente l'interface « *Decor* ». Cette classe définie toutes les méthodes de la classe « *Decor* ». Elle représente un décor qui ne se déplace vers aucune direction, mais qui possède son propre thread.

Au sein de cette classe, nous avons défini les attributs suivants :

- ⇨ « *etat* », qui représente l'état du décor (c'est-à-dire l'indice de l'image affichée) ;
- ⇨ « *positionX* », qui représente la position en abscisse du décor ;
- ⇨ « *positionY* », qui représente la position en ordonnée du décor ;
- ⇨ « *images* », qui représente une liste d'images permettant le stockage des images du décor.

Les attributs « *positionX* » et « *positionY* » sont des attributs possédant le mot-clé « *protected* » afin de pouvoir les modifier au sein de la classe héritant de celle-ci.

Le constructeur de cette classe prend en argument le chemin à emprunter pour récupérer les images, le nombre d'images à récupérer et enfin la condition de fin de partie, afin de stopper le thread. Dans le constructeur, nous faisons ensuite appel à la méthode « *loadImages* » afin de stocker l'ensemble des images au sein de l'attribut « *images* ».

Comme cette classe hérite de la classe « *Thread* », nous redéfinissons également la méthode « *run* » afin que, tant que la partie n'est pas terminée, l'image affichée est modifiée, via la méthode « *setEtat* ».

Afin de représenter des décors différents, nous avons deux classes héritant de la classe « *DecorImmobile* », nommées « *Choji* » et « *Gai* » (dont les noms correspondent aux personnages de « *Naruto Shippuden* »). Au sein du constructeur de ces classes, nous utilisons le mot-clé « *super* » afin de faire appel au constructeur de la classe « *DecorImmobile* » en prenant en arguments des valeurs prédéfinies telles que le chemin vers les images, qui correspond à « *decors/choji/choji* » pour la classe « *Choji* » et le nombre d'images qui correspond à la constante « *NB_IMAGES* ».

Les attributs « *positionX* » et « *positionY* » sont ensuite modifiées arbitrairement afin de les positionner au niveau de la meilleure façon qui soit au niveau de l'horizon.

Afin de représente un décor mobile, nous avons créé la classe abstraite « *DecorMobile* » qui hérite de la classe « *Thread* » et implémente l'interface « *Decor* ». Cette classe contient les mêmes attributs de la classe « *DecorImmobile* », qui possèdent les mêmes utilités, aux seules différences que nous avons retiré l'attribut « *images* » afin de la remplacer par deux nouveaux attributs nommés « *imagesG* » et « *imagesD* » qui correspondent respectivement à l'image du décor se déplaçant vers la gauche et la droite, et que nous avons ajouté l'attribut « *isMovingToRight* », qui permet de déterminer vers quelle direction doit se déplacer le décor.

En plus des méthodes réécrites de l'interface « *Decor* », nous avons ajouté les méthodes :

- « *setIsMovingToRight* », qui modifie le côté vers lequel le décor se déplace ;
- « *moveToRight* », qui incrémente la position en abscisse du décors de la valeur de *PAS* ;
- « *moveToLeft* », qui décrémente la position en abscisse du décor de la valeur de *PAS* ;
- « *isRemovable* », qui permet de déterminer lorsque le décor est sorti de la fenêtre.

A la différence de « *loadImages* » de la classe « *DecorImmobile* », dans cette classe, nous récupérons à la fois les images du décor se déplaçant vers la gauche et les images du décor se déplaçant vers la droite.

Nous avons ensuite créé la classe héritant de celle-ci se nommant « *DeidaraTobi* » (représentant également les noms de deux personnages de « *Naruto Shippuden* »). Cette classe fonctionne de la même manière que les classes « *Choji* » ou « *Gai* », à l'exception que nous réécrivons les méthodes « *isRemovable* » et « *run* ».

Dans la méthode « *run* », nous faisons en sorte que, tant que la partie n'est pas terminée, l'image affichée du décor est modifiée via la méthode « *setEtat* » et l'image se déplace vers un côté correspondant à la variable « *isMovingToRight* » à l'aide des méthodes « *moveToRight* » ou « *moveToLeft* » jusqu'à ce que le décor sorte de la fenêtre, à ce moment, le décor se déplacera de l'autre côté. Ainsi, lorsque le décor quitte la fenêtre par la droite, il réapparaîtra par la droite et continuera son chemin jusqu'à gauche et une fois qu'il quittera la fenêtre par la gauche, il réapparaître consécutivement de la même manière.

Afin d'instancier ces décors, nous avons créé la classe « *EtatDecorHorizon* ». Dans celle-ci, nous avons créé un attribut « *decors* » qui correspond à une liste de décors qui sera affiché sur l'horizon. A l'aide de la méthode « *fillListe* », nous remplissons cette liste de trois décors : « *Choji* », « *Gai* » et « *DeidaraTobi* ». Pour récupérer les éléments de cette liste sous forme de tableau qui permettra ainsi l'affichage des images, nous avons également ajouté la méthode « *getDecorsHorizon* ».

Afin d'afficher ces décors, nous avons créé la classe « *VueDecor* » et en son sein, la méthode « *drawDecorsHorizon* ». Dans celle-ci, nous récupérons les éléments de décors à l'aide de la méthode « *getDecorsHorizon* » de la classe « *EtatDecorHorizon* » et effectuons une boucle en faisant appel à la méthode « *drawImage* » de la classe « *Graphics* » en parcourant chaque élément du tableau afin les afficher. Cette méthode est ensuite appelée au sein de la classe « *Affichage* » dans la méthode « *paint* ».

DESSIN DE LA PISTE

Le dessin de la piste a nécessité la création de la classe « *Piste* ». Celle-ci consiste à créer des points aléatoires qui constitueront la piste. Nous avons tout d'abord ajouté au sein de cette classe les deux attributs suivants :

- « *liste* » de la classe « *ArrayList<Point>* », qui contiendra tous les points qui seront définis au sein de cette classe et qui seront nécessaires au traçage de la piste ;
- « *rand* » de la classe « *Random* », qui permet le remplissage des points à l'aide de valeurs aléatoires.

Nous y avons ensuite ajouté les constantes suivantes, nécessaires pour la prise de valeurs de points :

- « *LARGEUR_MIN* », la coordonnée en abscisse la plus petite que puissent prendre les points ;
- « *LARGEUR_MAX* », la coordonnée en abscisse la plus grande que puissent prendre les points ;
- « *HAUTEUR_MIN* », la coordonnée en ordonnée la plus petite que puissent prendre les points ;
- « *HAUTEUR_MAX* », la coordonnée en ordonnée la plus grande que puissent prendre les points ;
- « *ESPACEMENT_HAUTEUR* », la valeur à décrémenter à la coordonnée en ordonnée.

Ces constantes sont définies dans le but de délimiter la valeur que peuvent prendre, au sein du composant, les points qui constitueront la piste afin que celui-ci ne dépasse pas la ligne de l'horizon et de la fenêtre.

Ensuite, nous avons créé une méthode « *fillListe* » appelée au sein du constructeur de cette classe afin de donner une première liste de points. Dans cette méthode, nous effectuons une boucle qui remplit l'attribut « *liste* » de points.

Dans le cas de la coordonnée en abscisse des points, les valeurs sont choisies de manière aléatoire mais toujours en fonction des constantes précédemment citées et des méthodes « *getXMin* » et « *getXMax* » permettant de rester au sein de la zone où peut se positionner la piste.

Dans le cas de la coordonnée en ordonnée des points, les valeurs ne sont pas choisies aléatoirement : elles commencent à partir de la bordure inférieure de la fenêtre et remontent de manière constante en la décrémentant de la constante « *ESPACEMENT_Y* » qui correspond à un cinquième de la distance entre la bordure inférieure de la fenêtre et l'horizon). Elle permet ainsi de donnée à la piste un aspect de ligne brisée.

Nous ajoutons ensuite un accesseur nommé « *getPiste* » qui va parcourir chaque point de l'attribut « *liste* » et va les stocker au sein d'un tableau que la méthode retournera afin de récupérer les points qui constitueront la piste au cours de l'affichage.

Afin de l'afficher, nous avons créé la classe « *VuePiste* » disposant de l'attribut « *texture* ». Dans son constructeur, nous initialisons « *texture* » afin de stocker l'image qui représentera la texture de la piste (voir ci-dessous) :

Nous avons ensuite ajouté la méthode « *drawPiste* » dans laquelle nous récupérons les points de la piste via la méthode « *getPiste* » de la classe « *Piste* ». Afin de tracer la piste, nous optons pour la classe « *Path2D* ». Au pinceau, nous associons l'attribut « *texture* » à l'aide de la méthode « *setPaint* » et modifions son épaisseur à l'aide de la méthode « *setStroke* ». Nous effectuons ensuite une boucle en fonction du nombre de points du tableau récupéré. Dans cette boucle, afin de tracer la piste de manière continue, nous avons besoin de garder chaque second point qui constitue une ligne au sein d'une variable que nous utilisons ensuite afin de tracer la suite de la ligne à l'aide des méthodes « *moveTo* » et « *lineTo* » de la classe « *Path2D* ». Nous dessinons ensuite cette ligne brisée à l'aide de la méthode « *draw* ». La méthode « *drawPiste* » est ensuite appelée au sein de la méthode « *paint* ».

ANIMATION DE LA PISTE

L'animation de la piste a nécessité l'ajout d'une variable nommée « *positionY* » et d'accesseurs qui lui sont associés : « *getPositionY* », qui permet de récupérer la valeur de cette variable et « *setPositionY* », qui permet de l'incrémenter de la valeur de la constante *DEPLACEMENT* (qui correspond à un huitième de la coordonnée en ordonnée de deux points). Cette variable permet de décaler les points appartenant à la piste de la valeur de cette variable afin de simuler l'avancement du personnage le long de la piste.

L'appel à la méthode « *setPosition* » a été effectué au sein d'une nouvelle classe que nous avons créée, nommée « *Avancer* ». Cette classe hérite de la classe « *Thread* » et prend en paramètre un objet de la classe « *Piste* » afin d'utiliser cette dernière méthode. Ainsi, après un certain temps écoulé, la valeur de l'attribut « *positionY* » est incrémentée.

Le changement de valeur de cette variable prend effet au sein de la classe « *Affichage* » à laquelle nous avons ajouté à chaque coordonnée en ordonnée de la piste la valeur de cette variable que nous avons récupéré grâce à la méthode « *getPositionY* ». C'est ainsi que l'effet de déplacement de la piste se produit.

Cependant, il a été nécessaire de supprimer les points ne figurant plus sur la fenêtre, nous avons donc créé une méthode « *removePoint* » qui consiste, lorsque le premier point de la liste (c'est-à-dire le point le plus proche de la bordure inférieure de la fenêtre) se trouve à l'extérieur de la fenêtre, de la supprimer de la liste. Cette méthode est appelée au sein de la méthode « *getPiste* », avant de convertir la liste en tableau.

Également, il nous fallait prévoir de nouveaux points à ajouter au sein de la liste lorsque le dernier point de la liste (le point le plus proche de la bordure supérieure de la fenêtre) est égale ou inférieur à l'horizon, nous avons donc créé la méthode « *addPoint* » qui répond à ce besoin. Celle-ci est également appelée avant de récupérer le tableau de points correspondant à la piste grâce à la méthode « *getPiste* ».

PERSONNAGE ET ANIMATION

Une fois les images de notre personnage récupérées :

Nous avons créé une classe « *EtatNaruto* » qui hérite de la classe « *Thread* » et qui consiste à modifier les positions du personnage. Dans cette classe, nous définissons les constantes suivantes :

- « *NB_IMG* », le nombre d'images que nous avons du personnage ;
- « *LARGEUR_IMG* », la largeur de l'image du personnage ;
- « *HAUTEUR_IMG* », la hauteur de l'image du personnage ;
- « *POSITION_INITIALE_X* », la position en abscisse du personnage ;
- « *POSITION_INITIALE_Y* », la position en ordonnée du personnage.

Nous définissons également l'attribut « *etat* » qui va nous permettre de savoir dans quel état se trouve actuellement le personnage. Accessoirement, nous ajoutons au sein de la classe deux accesseurs : « *getEtat* », qui retourne l'état actuel du personnage et « *setEtat* », qui incrémente la valeur de « *etat* ».

Nous réécrivons ensuite la méthode « *run* » dans laquelle nous effectuons une boucle infinie et dans laquelle est appelée « *setEtat* » et qui consiste, après un certain délai, à modifier l'état du personnage.

Nous avons ensuite créé une nouvelle classe nommée « *VueNaruto* » qui permettra l'affichage du personnage. Dans celle-ci, nous définissons un attribut de la classe « *ArrayList<Image* » nommée « *middle* » que nous remplissons à l'aide de la méthode « *fillList* ». Cette dernière permet, via une boucle, de remplir d'images l'attribut « *middle* » afin que nous n'ayons pas toujours à les récupérer.

Une autre méthode a également été créée nommée « *drawNaruto* » qui appelle la méthode « *drawImage* » de la classe « *Graphics* » et prend en argument les constantes précédemment citées et l'attribut « *middle* » ayant pour indice l'état récupéré à partir de la méthode « *getEtat* » de la classe « *EtatNaruto* ». Cette méthode est ensuite appelée au sein de la classe « *Affichage* », dans la méthode « *paint* ».

DEPLACEMENT DU PERSONNAGE (DROITE ET GAUCHE)

Le déplacement du personnage a nécessité la modification de la classe « *EtatNaruto* ». Dans celle-ci, nous avons tout d'abord ajouté les constantes suivantes :

- « *LARGEUR_MIN* », la valeur la plus petite qu'atteint le personnage avant de sortir de la fenêtre ;
- « *LARGEUR_MAX* », la valeur la plus grande qu'atteint le personnage avant de sortir de la fenêtre ;
- « *MOUVEMENT* », la valeur incrémentée ou décrémentée en fonction du déplacement du personnage.

Nous lui avons également ajouté un attribut nommé « *positionX* » que nous avons initialisé à la valeur de la constante « *POSITION_INITIALE_X* ». Cet attribut permet de savoir l'emplacement du personnage sur l'axe des abscisses puisque celui-ci va varier lors du déplacement du personnage.

Les méthodes ajoutées sont les suivantes :

- *getPosition*, qui permet de récupérer la position actuelle du personnage ;
- *moveToRight*, qui permet de déplacer le personnage vers la droite (donc d'incrémenter l'attribut « *positionX* ») dans le cas où il ne dépasse pas la valeur de la constante « *LARGEUR_MAX* » ;
- *moveToLeft*, qui permet de déplacer le personnage vers la gauche (donc de décrémenter l'attribut « *positionX* ») dans le cas où il n'est pas inférieur à la valeur de la constante « *LARGEUR_MIN* ».

Cette fonctionnalité a également nécessité la création de la classe « *ControlPersonnage* », qui implémente la classe « *KeyListeneer* ». Celle-ci va provoquer, en fonction des touches appuyées ou pressées, le déplacement du personnage vers la direction souhaitée.

Nous avons donc réécrit les méthodes « *keyPressed* » et « *keyTyped* » afin que, lorsque l'utilisateur tape ou laisse enfoncer la touche directionnelle droite ou gauche, le personnage se déplace vers la direction adéquate via les méthodes « *moveToLeft* » ou « *moveToRight* ».

De plus, afin d'avoir une fluidité dans les mouvements du personnage, nous avons ajouté les attributs publiques « *left* » et « *right* » au sein de la classe « *EtatNaruto* » qui permettent respectivement de déterminer lorsque le personnage se déplace vers la gauche et lorsqu'il se déplace vers la droite. La valeur de ces attributs sont modifiés auprès de la classe « *ControlPersonnage* » en fonction des touches directionnelles appuyées/enfoncées.

VITESSE DU PERSONNAGE

La classe « *Vitesse* » représente la vitesse du personnage. Dans cette classe, nous définissons les constantes :

- « *VITESSE_INITIALE* », la vitesse initiale du personnage en début de partie ;
- « *VITESSE_MAX* », la valeur la plus grande que puisse prendre l'accélération du personnage ;
- « *VITESSE_MIN* », la valeur la plus petite que puisse prendre la décélération du personnage ;
- « *ACCELERE* », la valeur à incrémenter en cas d'accélération ;
- « *DECCELERE* », la valeur à décrémenter en cas de décélération.

Elle possède également l'attribut « *vitesse* » et l'accesseur « *getVitesse* », qui retourne la vitesse du personnage, ainsi que les méthodes « *accelerate* » et « *decceleration* », qui permettant respectivement d'incrémenter et de décrémenter la vitesse de la valeur des constantes « *ACCELERE* » et « *DECELERE* ». Puisqu'un des facteurs d'échec du joueur est la vitesse, nous avons également défini la méthode « *speedAtZero* » afin de déterminer lorsque la vitesse atteint zéro.

Les méthodes « *accelerate* » et « *decelerate* » sont utilisées au sein de la classe « *Piste* » lors de l'augmentation de la variable de décalage de la piste, au sein de la méthode « *setPositionY* », lorsque le personnage est positionné sur la piste (accélération) ou en dehors (décélération).

La vitesse est ensuite récupérée via la méthode « *getVitesse* » au sein de la classe « *Affichage* », dans laquelle nous avons défini une nouvelle méthode nommée « *drawVitesse* », qui permet d'afficher la vitesse via la méthode « *drawString* » de la classe « *Graphics* ». Cette méthode est ensuite appelée dans la méthode « *paint* ».

OBSTACLES ET DETECTION

L'ajout des obstacles a nécessité la recherche d'images correspondantes à l'univers de « *Naruto Shippuden* ». Nous avons donc opté pour les images suivantes :

Afin de représenter des obstacles, nous avons créé une classe « *Obstacle* » qui hérite de la classe « *Thread* » et implémente l'interface « *Decor* ». Cette classe possède les attributs :

- « *positionX* » qui représente la coordonnée en abscisse de l'obstacle ;
- « *positionY* » qui représente la coordonnée en ordonnée de l'obstacle ;
- « *images* » qui représente la liste d'images représentatives de l'onstacle ;
- « *etat* » qui représente l'image affichée ;
- « *onlyOneTime* » qui permet de limiter le nombre de décrémentation de la vitesse et du score lorsque l'obstacle touche le personnage la première fois.

Cette classe redéfinie l'ensemble des méthodes de la classe « *Decor* » et y ajouter la méthode « *isTouched* » qui permet de déterminer lorsque le personnage a touché une première fois l'ostacle en utilisant les positions en abscisse et en ordonnée du personnage via les méthodes « *getPositionX* » et la constante « *POSITION_INITIALE_Y* » ainsi que la taille des images du personnage de la classe « *EtatNaruto* ».

Cette classe redéfinie également la méthode « *run* » de la classe « *Thread* » afin que, tant que la partie n'est pas terminée, si le personnage est touché par un obstacle, le score et la vitesse sont décrémentés à l'aide de l'appel aux méthodes « *decreaseByObstacle* » de la classe « *Score* » et « *decelerate* » de la classe « *Vitesse* » et l'attribut « *onlyOneTime* » obtient la valeur « *false* » afin de n'effectuer cela qu'une unique fois.

Pour générer des obstacles aléatoires, nous avons créé la méthode « *EtatObstacle* ». Cette classe possède les constantes suivantes :

- « *NB_OBSTACLES* », le nombre d'obstacles différents ;
- « *NB_MAX_OBSTACLES* », le nombre d'obstacles sur la fenêtre ;
- « *ESPACEMENT_Y* », la valeur en ordonnée à laquelle un nouvel obstacle est ajouté.

Cette classe possède également les attributs « *liste* » qui représente une liste d'obstacles, « *rand* » un générateur de valeurs aléatoires et enfin « *ressources* » un tableau à deux dimensions qui contient les chemins vers les images des obstacles et le nombre d'images que possèdent chaque obstacle, qui est initialisé dans le constructeur.

Cette classe possèdent la méthode « *fillListe* », qui remplit la liste de premiers obstacles. Afin de choisir aléatoirement quelles images contiendra l'obstacle, nous avons créé la méthode « *generatePath* » qui retourne un tableau contenant le

chemin vers les images de l'obstacle et le nombre d'images qui correspond à cet obstacle qui est choisi aléatoirement en utilisant l'attribut « *rand* » qui génère une valeur aléatoire et qui est utilisé dans l'attribut « *ressources* ». Un obstacle est généré à chaque distance correspondante à la constante « *ESPACEMENT_Y* ».

Afin de récupérer ces obstacles, nous avons créé la méthode « *getObstacles* » qui permet de retourner, sous forme de tableau, les obstacles contenus dans la liste. Comme la position en ordonnée de l'obstacle dépend de la variable de décalage de la piste, il est important de retirer et d'ajouter au sein de la liste de nouveaux obstacles. Ainsi, dans la méthode « *getObstacle* », nous avons fait appel aux méthode « *removeObstacle* », qui permet de retirer le premier obstacle de la liste lorsque sa coordonnée en ordonnée dépasse la hauteur de la fenêtre, et « *addObstacle* » qui permet d'ajouter un nouvel obstacle choisi aléatoirement via la méthode « *generatePath* » au sein de la liste lorsque la position en ordonnée du dernier obstacle est supérieur à la constante « *ESPACEMENT_Y* ».

Afin d'afficher ces obstacles, nous avons créé la classe « *VueObstacle* » dans laquelle nous avons créé la méthode « *drawObstacles* » qui permet de récupérer, via la méthode « *getObstacles* » de la classe « *EtatObstacle* » la liste d'obstacles sous forme de tableau et on effectue une boucle sur la méthode « *drawImage* » de la classe « *Graphics* » afin de dessiner l'ensemble des obstacles contenu dans le tableau récupéré.

Cette méthode de la classe « *VueObstacle* » est ensuite appelée au sein de la méthode réécrite « paint » de la classe « *Affichage* ».

POINT DE CONTROLE

Le point de contrôle a été représenté par un personnage appartenant à l'univers de « *Naruto Shippuden* » que voici :

Pour représenter le point de contrôle, nous avons créé la classe « *PointControle* » qui hérite de la classe « *Thread* » et implémente l'interface « *Decor* ». Cette classe possède une constante « *NB_IMAGES* » qui est le nombre d'images que l'on possède du point de contrôle. Elle possède également des attributs telles que : « *positionX* » et « *positionY* » qui représentent respectivement les positions en abscisse et en ordonnée du point de contrôle, « *etat* » qui représente l'état du point de contrôle, « *images* » qui est une liste contenant des images du point de contrôle et enfin « *onlyOneTime* » qui permet d'incrémenter la vitesse et le score qu'une unique fois par point de contrôle atteint.

Dans cette classe sont réécrites les méthodes de la classe « *Decor* » auxquelles nous avons ajouté la méthode « *isReached* » qui permet de déterminer lorsque le personnage a atteint le point de contrôle. Pour cela, on compare la position en ordonnée du point de contrôle et du personnage et si la position en ordonnée du personnage est supérieure à celle du point de contrôle, alors « *true* » est retournée.

Cette méthode est utilisée ensuite au sein de la méthode héritée « *run* » de la classe « *Thread* » afin que, lorsque le personnage a atteint le point de contrôle, le score et le décompte sont incrémentés respectivement via les méthodes « *increaseByCheckpoint* » de la classe « *Score* » et « *increaseTime* » de la classe « *Decompte* ».

Le point de contrôle est géré à partir de la classe « *EtatPointControle* » qui hérite de la classe « *Thread* », dans laquelle on définit les constantes suivantes :

- ⇨ « *LARGEUR_IMG* », la largeur de l'image ;
- ⇨ « *HAUTEUR_IMG* », la hauteur de l'image.

Les attributs définis dans cette classe sont « *seuil* » qui correspond à la valeur à laquelle apparaîtra le point de contrôle, « *pointControle* » qui correspond au point de contrôle et enfin « *isReached* » qui permet de déterminer lorsque le point de contrôle est atteint.

Nous avons ensuite défini les méthodes « *increaseLimit* » qui permet d'incrémenter de la valeur de la constante *SEUIL*, l'attribut « *seuil* », « *getCheckpoint* » qui retourne le point de contrôle et enfin « *spawnCheckpoint* » qui permet de faire apparaître le point de contrôle à partir du moment où le score a atteint la valeur du seuil. A ce moment-là, la valeur de seuil est incrémentée via la méthode « *increaseLimit* ».

Cette méthode est appelée au sein de la méthode héritée « *run* » afin que, tant que la partie n'est pas terminée, si la valeur de seuil est atteinte par le score, alors le point de contrôle apparaît.

La méthode « *drawCheckpoint* » permettant l'affichage du point de contrôle est ensuite défini au sein de la classe « *VuePointControle* » dans laquelle on récupère le point de contrôle via la méthode « *getCheckpoint* » de la classe « *EtatPointControle* » puis on l'affiche à l'aide de la méthode « *drawImage* » de la classe « *Graphics* » et des méthodes présentes dans la classe « *PointControle* ».

PROFIL DU PERSONNAGE

Nous avons ajouté cette fonctionnalité afin de déterminer lorsque le personnage a reçu des dégâts ou lorsque celui-ci a atteint un point de contrôle. Dans ces deux cas, une image différente est affichée, sinon, une image par défaut est affichée, voici les images utilisées pour symboliser le profil du personnage :

Nous avons donc créé la classe « *EtatProfilNaruto* » qui hérite de la classe « *Thread* ». Dans cette classe, nous avons créé l'attribut « *etat* » qui correspond à l'état du profil. La méthode « *getEtat* » permet de le récupérer. Nous redéfinissons ensuite la méthode héritée « *run* » de la classe « *Thread* » afin que, tant que la partie n'est pas terminée, l'image du profil soit modifiée en fonction de la valeur de l'état. Ainsi, grâce à l'attribut « *isTouched* » de la classe « *EtatNaruto* » qui détermine lorsque le personnage est touché par un obstacle et « *isReached* » de la classe « *EtatPointControle* » qui détermine lorsque le personnage a atteint un point de contrôle, l'état est modifié via les méthodes « *stateIsHurt* » et « *stateIsHappy* ». Par défaut, « *stateIsNormal* » est appelée. Ainsi, grâce aux constantes « *PAUSE_SUR_IMAGE* », l'image affichée perdure pendant un certain temps avant de redevenir l'image par défaut.

Nous définissons la méthode « *drawProfilNaruto* » permettant d'afficher le profil du personnage dans la classe « *VueProfilNaruto* ». Nous commençons tout d'abord à récupérer les images du profil via la méthode « *loadImages* » que l'on stocke ensuite dans l'attribut « *images* » qui correspond à une liste d'images. Nous utilisons ensuite la méthode « *drawImage* » de la classe « *Graphics* » afin d'afficher l'image de l'attribut « *images* » via la méthode « *getEtat* » qui permet de récupérer l'état du profil du personnage.

Cette méthode « *drawProfilNaruto* » est ensuite appelée au sein de la classe « *Affichage* », dans la méthode héritée « *paint* ».

SCORE

La classe « *Score* » représente le score du joueur. Dans cette classe, nous définissons les constantes :

- « *TIME* », la valeur à incrémenter au fil du temps ;
- « *CHECKPOINT* », la valeur à incrémenter lorsque le personnage atteint un point de contrôle ;
- « *OPPONENT* », la valeur à incrémenter lorsque le personnage dépasse un adversaire ;
- « *OBSTACLE* », la valeur à décrémenter lorsque le personnage est touché par un obstacle.

Elle possède également l'attribut « score » qui représente le score du joueur qui débute à zéro.

Elle possède les méthodes « *getScore* » qui retourne le score du joueur, « *increaseByTime* » qui incrémente le score au fil du temps, « *increaseByCheckpoint* » qui incrémente le score lorsque le personnage atteint un point de contrôle, « *increaseByOpponent* » qui incrémente le score lorsque le personnage dépasse un adversaire et enfin la méthode « *decreaseByObstacle* » qui décrémente le score lorsque le personnage est touché par un obstacle.

Les méthodes « *increaseByTime* », « *increaseByCheckpoint* » et « *decreaseByObstacle* » sont ensuite appelées respectivement au sein des classes « *Avancer* », « *PointControle* » et « *Obstacle* ».

L'affichage du score est ensuite effectué au sein de la classe « *Affichage* », dans laquelle nous avons ajouté une méthode nommée « *drawScore* », qui récupère le score du joueur à l'aide de la méthode « *getScore* ». Cette méthode est ensuite appelée au sein de la méthode « *paint* » de « *Affichage* ».

FIN DE PARTIE

Pour déterminer la fin de partie, nous avons créé la classe « *FinPartie* » qui possède les attributs « *isClosing* » qui permet de déterminer lorsque la fenêtre de jeu se ferme avant d'avoir terminé la partie, « *nombrePointsControle* » qui représente le nombre de points de contrôle atteints par le personnage et « *nombreObstacles* » qui représente le nombre d'obstacles touchés par le personnage. Cette classe possède également les méthodes « *setIsClosing* » qui permet de déterminer que la fenêtre de jeu a été fermée avant que la partie ne se termine, « *increaseCheckpoints* » qui incrémente le nombre de nombre de points de contrôle, « *increaseObstacles* » qui incrémente le nombre d'obstacles, « *gameOver* » qui permet de déterminer la fin de partie lorsque la vitesse du personnage ou le temps restant atteint zéro et enfin « *showGameOver* » qui permet d'afficher le score du joueur lorsque la partie se termine correctement.

La méthode « *gameOver* » est appelée dans toutes les classes héritant de la classe « *Thread* » afin d'éviter les boucles infinies. Quant à la méthode « *showGameOver* », elle est appelée au sein de la classe « *Affichage* », au sein de la méthode « *run* » après la boucle permettant la mise à jour de l'affichage.

Quant aux méthodes « *increaseCheckpoint* » et « *increaseObstacles* », elles sont appelées respectivement au sein des classes « *PointControle* » et « *Obstacle* » lorsque le point de contrôle est atteint par le personnage et lorsque le personnage est touché par un obstacle.

ECRAN D'ACCUEIL

Nous avons tout d'abord créé la classe « *FenetreAccueil* » qui hérite de la classe « *JFrame* » et représente la fenêtre sur laquelle le composant représentant l'accueil sera affiché. Ensuite, nous avons ajouté à la fenêtre le composant qui correspond à la classe « *VueAccueil* », qui hérite de la classe « *JPanel* ». Dans cette classe, nous avons défini les attributs « *imageFond* » et « *imageCurseur* » qui représentent respectivement l'image en fond d'écran du composant et l'image du curseur que nous initialisons via la méthode « *loadImages* ». Nous avons opté pour les images suivantes :

Afin d'afficher l'image de l'attribut « *imageFond* », nous redéfinissons la méthode hérité « *paint* » dans laquelle nous appelons, via la classe « *Graphics* », la méthode « *drawImage* » sur cet attribut.

Nous avons également défini trois autres méthodes : « *initializeButtons* », « *disableOptions* » et « *setGridBagConstraints* » qui permettent d'effectuer une mise en page permettant d'aligner deux boutons par lignes : l'une contenant l'image du curseur et l'autre contenant l'intitulé du bouton.

Sur les boutons possédant un intitulé, nous ajoutons, via la méthode « *addMouseListener* », un objet de la classe « *ControlSouris* » qui implémente l'interface « *MouseListener* », qui récupère le bouton sur lequel sera affiché l'image du curseur et l'image du curseur. Dans cette classe, nous redéfinissons les méthodes « *mouseEntered* » afin que lorsque le bouton ayant un intitulé soit survolé, le bouton près du bouton survolé affichera l'image du curseur. Et la méthode « *mouseExited* » qui retire l'image du curseur lorsque le composant associé n'est plus survolé.

Au bouton possédant un intitulé, nous ajoutons également, via la méthode « *addActionListener* », un objet de la classe « *ControlAccueil* » qui implémente l'interface « *ActionListener* » afin que lorsque le bouton est cliqué, une méthode spécifique soit effectuée. Les méthodes sont effectuées en fonction de l'intitulé du bouton et appartiennent à la classe « *EtatAccueil* ».

Cette classe possède des méthodes nommées « *start* » qui permet d'ouvrir une fenêtre de dialogue et qui, lorsque le bouton « *ok* » est entré, ouvre la fenêtre de jeu, « *score* » qui permet d'afficher les scores et enfin « *quit* » qui permet de quitter le programme.

Elle possède également la méthode « *setFrame* » qui permet de récupérer la fenêtre à fermer ou à rendre invisible. Cette méthode est appelée au sein de la classe « *FenetreAccueil* » afin de récupérer la fenêtre d'accueil.

Cette méthode est importante puisqu'elle permet de rendre invisible la fenêtre d'accueil lorsque la fenêtre de jeu est affichée. Nous avions ensuite besoin de revenir sur la fenêtre d'accueil via la fenêtre de jeu. Pour que cela se produise, nous avons créé la classe « *ControlFenetrePrincipale* » qui implémente l'interface « *WindowListener* », que nous ajoutons, via la méthode « *addWindowListener* », au sein de la classe « *FenetrePrincipale* ».

Ainsi, nous redéfinissons la méthode « *windowClosing* » afin de fermer la fenêtre de jeu via la méthode « *dispose* » et de rendre visible la fenêtre d'accueil via la méthode « *setVisible* ». Afin de fermer correctement la fenêtre de jeu, nous avons également fait appel à la méthode « *setIsClosing* » de la classe « *FinPartie* » qui permet de si la fermeture de la fenêtre a été brusquement interrompue, dans ce cas-là, tout le programme s'arrêtera correctement.

SONS

Différents audios sont mis à disposition selon les évènements provoqués par le joueur:

- Quand il cogne un obstacle ;
- Quand il depasse un point de contrôle ;
- Quand il perd la partie.

La classe « *AudioJeu* » utilise une variable de type « *Clip* » de la bibliothèque « *import javax.sound.sampled.Clip* ».

« *Clip* » représente un type de ligne de donnée dont les audios sont chargés avant la lecture, au lieu d'être diffusés en temps réel. Les données sont préchargées et ont une longueur connue.

La lecture de l'audio est démarée et arrêtée grâce aux méthodes « *start* » et « *stop* ».

« *AudioInputStream* » permet d'obtenir le flux audio à partir du fichier fournit et « open » ouvre le clip avec le format et des données audio présents dans le flux d'entrée fourni.

MEMORISATION DES 10 MEILLEURS SCORES

« *HighScore* » contiendra les 10 meilleures scores enregistés dans un fichier nommé « *MeilleureScore.txt* ».

le constructeur crée un meilleure score et essaie de le charger à partir d'un fichier utilisant un délimiteur donné. Le meilleure score est borné par un nombre maximum (dans notre cas : 10).

On crée un meilleure score à 2 colonnes : « *nom* » et « *score* » qui seront triés d'abord sur la deuxième colonne (colonne « *score* ») de manière décroissante et si on a deux scores égaux, alors le trie sera fait sur la première colonne (colonne « *nom* ») de manière croissante.

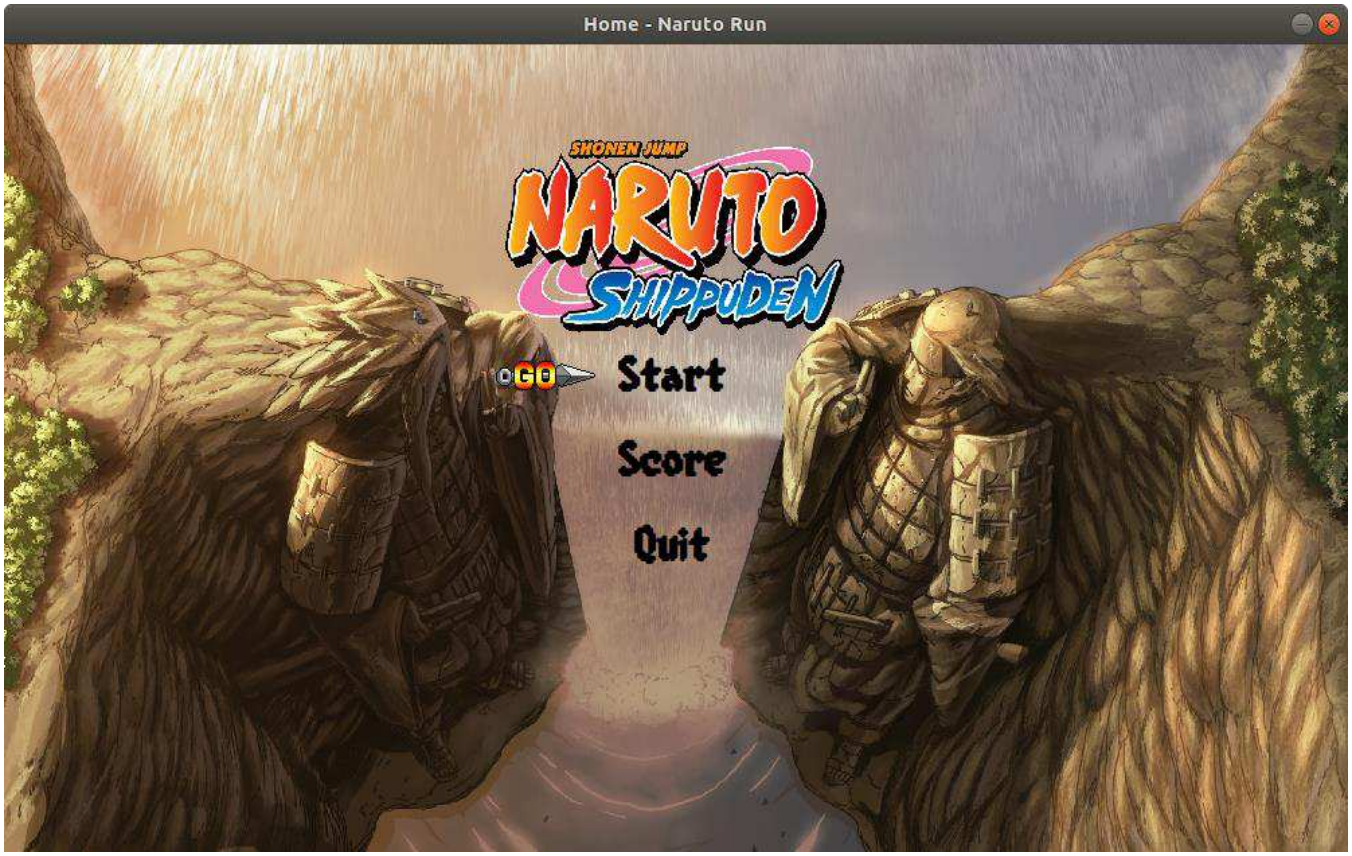
L'interface publique « *Comparable <T>* » impose un ordre sur les objets de chaque classe qui l'implémente. Cet ordre est appelé ordre naturel de la classe et la méthode « *compareTo* » de la classe est appelée la méthode de comparaison naturelle.

La « *Vector<T>* » implémente un tableau d'objets évolutifs. Comme un tableau, il contient des composants accessibles à l'aide d'un index.

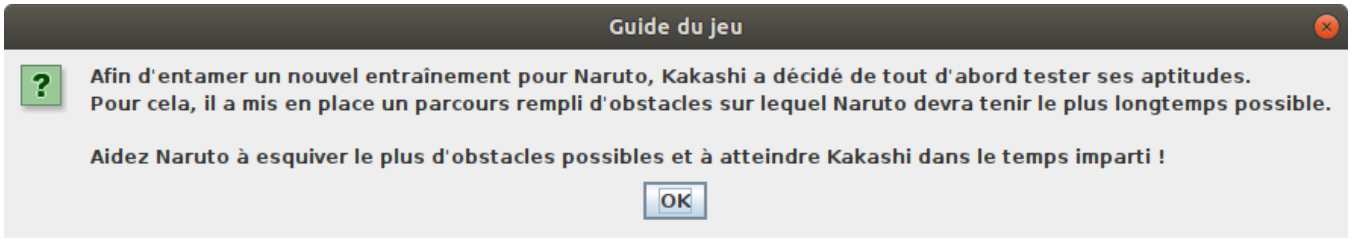
On utilise un tableau de comparaison qui s'occupe des differentes comparaisons à effectuer sur les colonnes du fichier « *MeilleureScore.txt* ».

Résultat

ECRAN D'ACCUEIL



BOUTON « START » : GUIDE DU JEU

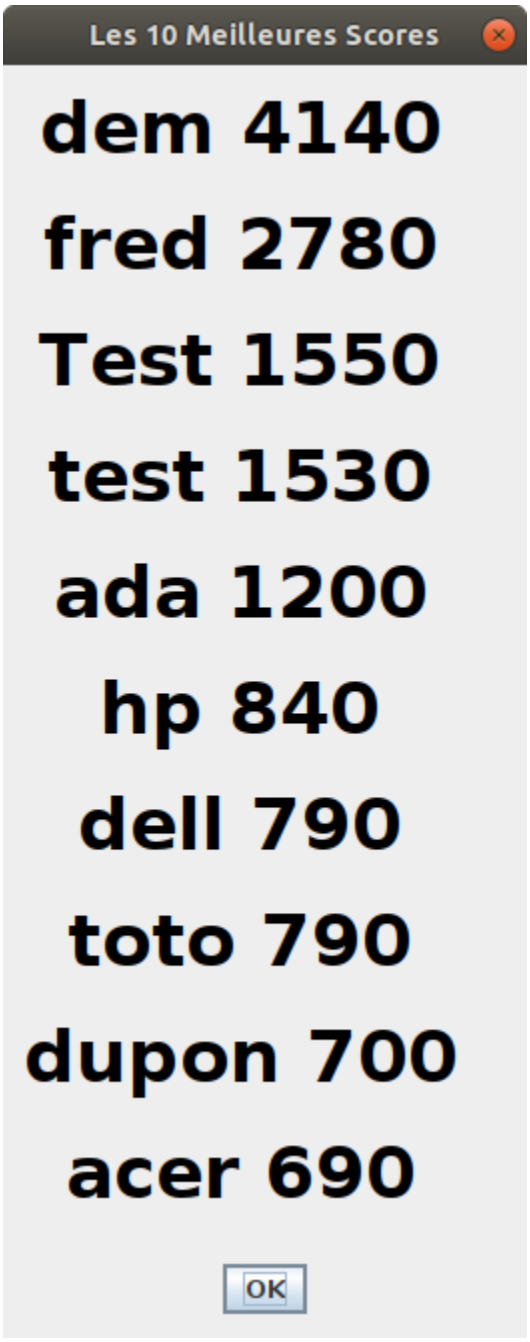


BOUTON « START » : AFFICHAGE DU JEU

FIN DE PARTIE : ENREGISTRER SON SCORE

FIN DE PARTIE : DETAILS DU SCORE OBTENU

BOUTON « SCORE » : AFFICHAGE DES 10 MEILLEURS SCORES



Conclusions personnelles

JANETHE FERHOUNE

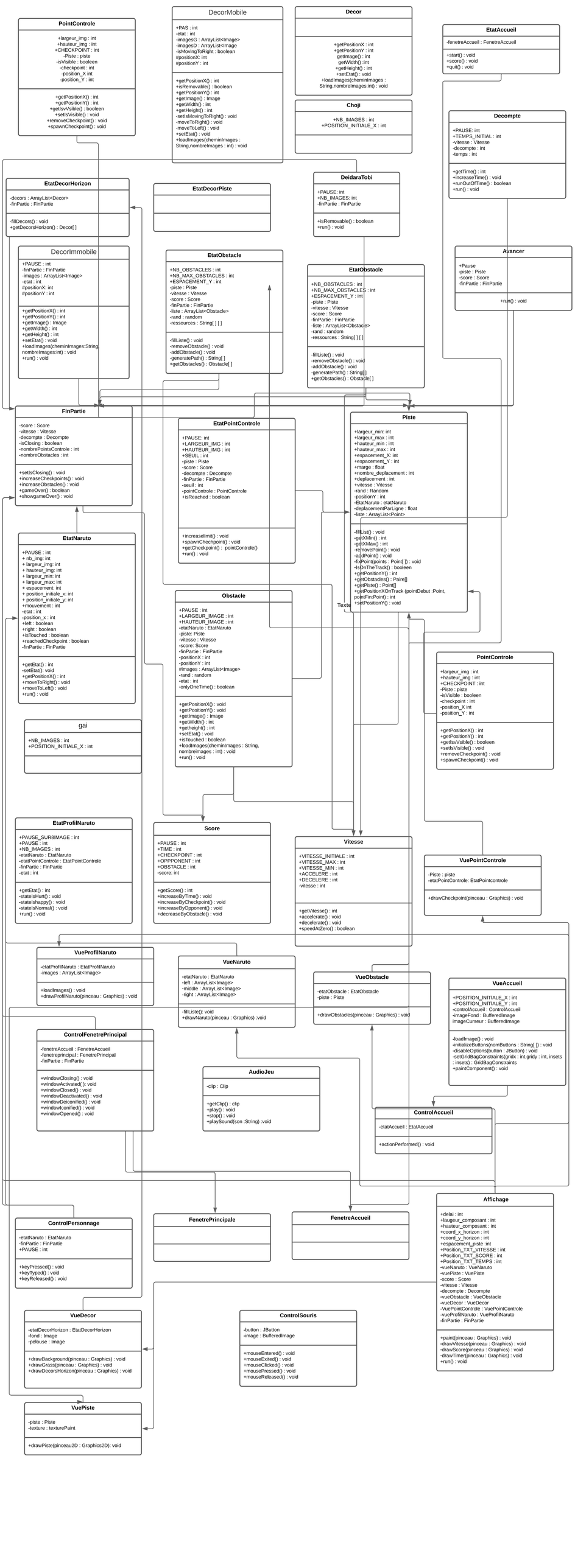
Pour ma part, ce projet m'a été très instructif, aussi bien au niveau de la mise en pratique de mes connaissances qu'au niveau du travail en équipe, qui est une partie importante pour le bon déroulement d'un projet. En effet, la répartition du travail et la confiance en son coéquipier sont primordiaux. C'est pourquoi je pense que notre groupe avait un bon rythme de progrès, notamment grâce à notre bonne entente, mais également grâce à l'intérêt que nous portions pour ce projet, qui est, me concernant, un sujet qui m'a beaucoup inspiré. Ainsi, je pense pouvoir dire que ce projet est réussi, malgré une gestion du temps beaucoup trop juste et malgré le fait que nous n'avions pas eu le temps de terminer les dernières fonctionnalités telles que l'ajout d'adversaires. Notre équipe s'est entraidée au fur et à mesure de notre avancement, malgré des conditions en distanciel assez désastreuses, notamment dues à nos mauvaises connexions. J'ai appris beaucoup de ce projet, notamment dans l'utilisation de la classe « *Thread* » et d'interfaces. Cependant, il est vrai que j'aurais apprécié user davantage de la classe « *Graphics2D* » lors de notre projet, c'est pourquoi je pense à présent me renseigner afin de découvrir toute l'étendue de ses capacités. Ce projet fut amusant, bien que stressant à bien des points et a réussi à attiser ma curiosité de bien des manières puisque, jusqu'à la réalisation de ce projet, je ne me croyais pas capable de réaliser un jeu vidéo. Pour finir, la partie dont je suis la plus fière n'est autre que l'animation du personnage avec les sprites de directions adéquates : c'était la première fois que j'utilisais un thread de cette manière, ainsi l'ajout de cette fonctionnalité m'a amené à une nouvelle vision du jeu, c'est ainsi qu'est né le thème de « *Naruto Shippuden* ».

HAWA INA OURO DIALLO

…

Annexe(s)

DIAGRAMME DE CLASSE


PointControle
+largeur_img : int
+hauteur_img : int
+CHECKPOINT : int
-Piste : piste
-isVisible : booleen
-checkpoint : int
-position_X int
-position_Y : int
+getPositionX() : int
+getPositionY() : int
+getIsvVisible() : booleen
+setIsVisible() : void
+removeCheckpoint() : void
+spawnCheckpoint() : void
DecorMobile
+PAS : int
-etat : int
-imagesG : ArrayList<Image>
-imagesD : ArrayList<Image
-isMovingToRight : boolean
#positionX: int
#positionY : int
+getPositionX() : int
+isRemovable() : boolean
+getPosition(Y() : int
+getImage() : Image
+getWidth() : int
+getHeight() : int
-setIsMovingToRight() : void
-moveToRight() : void
-moveToLeft() : void
+setEtat() : void
+loadImages(cheminImages :
String,nombreImages : int) : void
Decor
+getPositionX : int
+getPositionY : int
getImage() : int
getWidth() :int
+getHeight() : int
+setEtat() : void
+loadImages(cheminImages :
String,nombreImages:int) : void
EtatAccueil
-fenetreAccueil : FenetreAccueil
+start() : void
+score() : void
+quit() : void
Choji
+NB_IMAGES : int
+POSITION_INITIALE_X : int
Decompte
+PAUSE: int
+TEMPS_INITIAL : int
-vitesse : Vitesse
-decompte : int
-temps : int
+getTime() : int
+increaseTime() : void
+runOutOfTime() : boolean
+run() : void
EtatDecorHorizon
-decors : ArrayList<Decor>
-finPartie : FinPartie
-fillDecors() : void
+getDecorsHorizon() : Decor[ ]
EtatDecorPiste
DeidaraTobi
+PAUSE: int
+NB_IMAGES: int
-finPartie : FinPartie
+isRemovable() : boolean
+run() : void
DecorImmobile
+PAUSE : int
-finPartie : FinPartie
-images : ArrayList<Image>
-etat : int
#positionX: int
#positionY : int
+getPositionX() : int
+getPositionY() : int
+getImage() : Image
+getWidth() : int
+getHeight() : int
+setEtat() : void
+loadImages(cheminImages:String,
nombreImages:int) : void
+run() : void
EtatObstacle
+NB_OBSTACLES : int
+NB_MAX_OBSTACLES : int
+ESPACEMENT_Y : int
-piste : Piste
-vitesse : Vitesse
-score : Score
-finPartie : FinPartie
-liste : ArrayList<Obstacle>
-rand : random
-ressources : String[ ] [ ]
-fillListe() : void
-removeObstacle() : void
-addObstacle() : void
-generatePath() : String[ ]
+getObstacles() : Obstacle[ ]
EtatObstacle
+NB_OBSTACLES : int
+NB_MAX_OBSTACLES : int
+ESPACEMENT_Y : int
-piste : Piste
-vitesse : Vitesse
-score : Score
-finPartie : FinPartie
-liste : ArrayList<Obstacle>
-rand : random
-ressources : String[ ] [ ]
-fillListe() : void
-removeObstacle() : void
-addObstacle() : void
-generatePath() : String[ ]
+getObstacles() : Obstacle[ ]
Avancer
+Pause
-piste : Piste
-score : Score
-finPartie : FinPartie
+run() : void
FinPartie
-score : Score
-vitesse : Vitesse
-decompte : Decompte
-isClosing : boolean
-nombrePointsControle : int
-nombreObstacles : int
+setIsClosing() : void
+increaseCheckpoints() : void
+increaseObstacles() : void
+gameOver() : boolean
+showgameOver() : void
EtatPointControle
+PAUSE: int
+LARGEUR_IMG : int
+HAUTEUR_IMG : int
+SEUIL : int
-piste : Piste
-score : Score
-decompte : Decompte
-finPartie : FinPartie
-seuil : int
-pointControle : PointControle
+isReached : boolean
+increaseIimit() : void
+spawnChechpoint() : void
+getCheckpoint() : pointControle()
+run() : void
Piste
+largeur_min: int
+largeur_max : int
+hauteur_min : int
+hauteur_max : int
+espacement_X: int
+espacement_Y : int
+marge : float
+nombre_deplacement : int
+deplacement : int
+vitesse : Vitesse
-rand : Random
-positionY : int
-EtatNaruto : etatNaruto
-deplacementParLigne : float
-liste : ArrayList<Point>
-fillList() : void
-getXMin() : int
-getXMax() : int
-removePoint() : void
-addPoint() : void
-fixPoint(points : Point[ ]) : void
-IsOnTheTrack() : booleen
+getPositionY() : int
+getObstacles() : Paire[]
+getPiste() : Point[]
+getPositionXOnTrack (pointDebut :Point,
pointFin:Point) : int
+setPositionY() : void
EtatNaruto
+PAUSE : int
+ nb_img: int
+ largeur_img: int
+ hauteur_img: int
+ largeur_min: int
+ largeur_max: int
+ espacement: int
+ position_initiale_x: int
+ position_initiale_y: int
+mouvement : int
-etat : int
-position_x : int
+left : boolean
+right : boolean
+isTouched : boolean
+reachedCheckpoint : boolean
-finPartie : FinPartie
+getEtat() : int
-setEtat(): void
+getPositionX() : int
+moveToRight() : void
+moveToLeft() : void
+run() : void
Obstacle
+PAUSE : int
+LARGEUR_IMAGE : int
+HAUTEUR_IMAGE : int
-etatNaruto : EtatNaruto
-piste: Piste
-vitesse : Vitesse
-score: Score
-finPartie : FinPartie
-positionX : int
-positionY : int
#images : ArrayList<Image>
-rand : random
-etat : int
-onlyOneTime() : boolean
+getPositionX() : void
+getPositionY() : void
+getImage() : Image
+getWidth() : int
+getheight() : int
+setEtat() : void
+isTouched : boolean
+loadImages(cheminImages : String,
nombreimages : int) : void
+run() : void
Texte
PointControle
+largeur_img : int
+hauteur_img : int
+CHECKPOINT : int
-Piste : piste
-isVisible : booleen
-checkpoint : int
-position_X int
-position_Y : int
+getPositionX() : int
+getPositionY() : int
+getIsvVisible() : booleen
+setIsVisible() : void
+removeCheckpoint() : void
+spawnCheckpoint() : void
gai
+NB_IMAGES : int
+POSITION_INITIALE_X : int
EtatProfilNaruto
+PAUSE_SUR8IMAGE : int
+PAUSE : int
+NB_IMAGES : int
-etatNaruto : EtatNaruto
-etatPointControle : EtatPointControle
-finPartie : FinPartie
-etat : int
+getEtat() : int
-stateIsHurt() : void
-stateIshappy() : void
-stateIsNormal() : void
+run() : void
Score
+PAUSE : int
+TIME : int
+CHECKPOINT : int
+OPPPONENT : int
+OBSTACLE : int
-score: int
+getScore() : int
+increaseByTime() : void
+increaseByCheckpoint() : void
+increaseByOpponent() : void
+decreaseByObstacle() : void
Vitesse
+VITESSE_INITIALE : int
+VITESSE_MAX : int
+VITESSE_MIN : int
+ACCELERE : int
+DECELERE : int
-vitesse : int
+getVitesse() : int
+accelerate() : void
+decelerate() : void
+speedAtZero() : boolean
VuePointControle
-Piste : piste
-etatPointControle: EtatPointcontrole
+drawCheckpoint(pinceau : Graphics) : void
VueProfilNaruto
-etatProfilNaruto : EtatProfilNaruto
-images : ArrayList<Image>
+loadImages() : void
+drawProfilNaruto(pinceau : Graphics) : void
VueNaruto
-etatNaruto : EtatNaruto
-left : ArrayList<Image>
-middle : ArrayList<Image>
-right : ArrayList<Image>
-fillListe(): void
+drawNaruto(pinceau : Graphics) :void
VueObstacle
-etatObstacle : EtatObstacle
-piste : Piste
+drawObstacles(pinceau : Graphics) : void
VueAccueil
+POSITION_INITIALE_X : int
+POSITION_INITIALE_Y : int
-controlAccueil : ControlAccueil
-imageFond : BufferedImage
imageCurseur : BufferedImage
-loadImage() : void
-initializeButtons(nomButtons : String[ ]) : void
-disableOptions(button : JButton) : void
-setGridBagConstraints(gridx : int,gridy : int, insets
: insets) : GridBagConstraints
+paintComponent() : void
ControlFenetrePrincipal
-fenetreAccueil : FenetreAccueil
-fenetreprincipal : FenetrePrincipal
-finPartie : FinPartie
+windowClosing() : void
+windowActivated() : void
+windowClosed() : void
+windowDeactivated() : void
+windowDeiconified() : void
+windowIconified() : void
+windowOpened() : void
AudioJeu
-clip : Clip
+getClip() : clip
+play() : void
+stop() : void
+playSound(son :String) :void
ControlAccueil
-etatAccueil : EtatAccueil
+actionPerformed() : void
ControlPersonnage
-etatNaruto : EtatNaruto
-finPartie : FinPartie
+PAUSE : int
+keyPressed() : void
+keyTyped() : void
+keyReleased() : void
FenetrePrincipale
FenetreAccueil
Affichage
+delai : int
+laugeur_composant : int
+hauteur_composant : int
+coord_x_horizon : int
+coord_y_horizon : int
+espacement_piste :int
+Position_TXT_VITESSE : int
+Position_TXT_SCORE : int
+Position_TXT_TEMPS : int
-vueNaruto : VueNaruto
-vuePiste : VuePiste
-score : Score
-vitesse : Vitesse
-decompte : Decompte
-vueObstacle : VueObstacle
-vueDecor : VueDecor
-VuePointControle : VuePointControle
-vueProfilNaruto : VueProfilNaruto
-finPartie : FinPartie
+paint(pinceau : Graphics) : void
-drawVitesse(pinceau : Graphics) : void
-drawScore(pinceau : Graphics) : void
-drawTimer(pinceau : Graphics) : void
+run() : void
VueDecor
-etatDecorHorizon : EtatDecorHorizon
-fond : Image
-pelouse : Image
+drawBackground(pinceau : Graphics) : void
+drawGrass(pinceau : Graphics) : void
+drawDecorsHorizon(pinceau : Graphics) : void
ControlSouris
-button : JButton
-image : BufferedImage
+mouseEntered() : void
+mouseExited() : void
+mouseClicked() : void
+mousePressed() : void
+mouseReleased() : void
VuePiste
-piste : Piste
-texture : texturePaint
+drawPiste(pinceau2D : Graphics2D): void