TRALALERE

Notions de programmation

Avec exemples propres à GameCode

GameCode

Créer et programmer son
jeu vidéo

Bibliothèques Sans Frontières
Libraries Without Borders

INVESTISSEMENTS D'AVENIR

Caisse des Dépôts

l'Europe s'engage en Ile-de-France avec le FEDER

UNION EUROPÉENNE

V1.0

Auteur : Tralalere

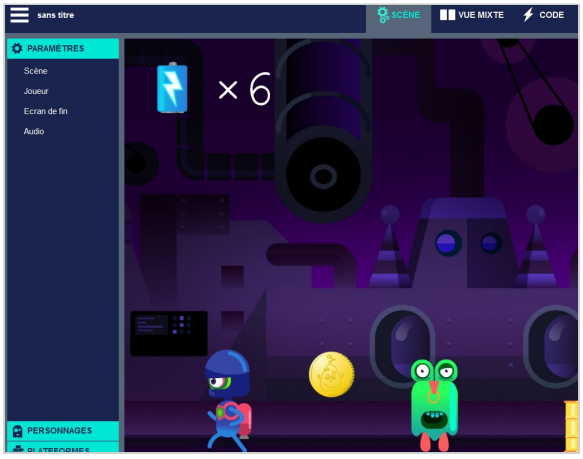
Sommaire

Introduction

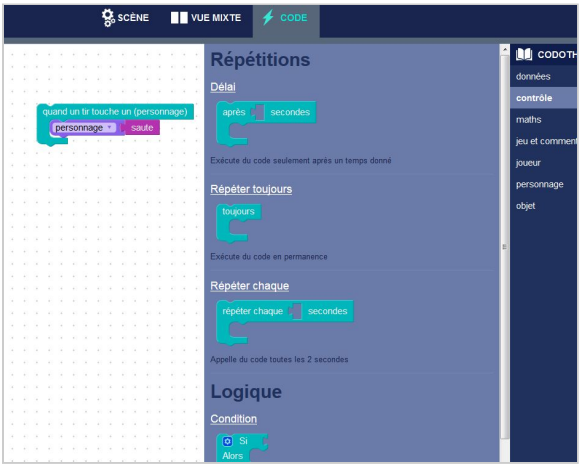
Ce document s'adresse aux acteurs éducatifs désireux de faire des ateliers de programmation avec l'application GameCode aussi bien en langage javascript qu'en bloc.

GameCode est un éditeur de jeu de plateforme composé de plusieurs zones :

Une zone de rendu permettant d'assembler les éléments de sa création



Une zone de script avec un accès à la codothèque



GameCode permet de s'initier au langage objet. À chaque type d'objet (joueur, personnage, bonus) peut être associé des variables, actions, événements et comportements particuliers. Ce document vous aidera à appréhender et comprendre le code de ces différentes caractéristiques pour ainsi mieux expliquer chaque ligne de code aux élèves.

Prérequis pour pouvoir utiliser ce guide :

- **Connaître l'application GameCode.** Nous vous invitons à découvrir l'application au travers du parcours « GAMECODE - TUTOS ».
- **Avoir des notions en programmation.** Nous vous invitons à consulter le document « Notions générales de programmation » disponible sur le site.

Pour approfondir :

- Découvrez de nombreux exemples de code au travers des documents « GameCode - blocs : exemples de code » et « GameCode - script : exemples de code » disponibles sur le site. Ces documents vous donneront des idées de comportements pour aider les élèves à concevoir un jeu lors d'ateliers.

Les éléments de scène

Dans GameCode, l'utilisateur va créer son environnement de jeu, il s'agit du niveau dans lequel le joueur va évoluer.

La scène a ses propres caractéristiques.

Elles permettent de définir :

- la valeur de la variable « gravité ». Il s'agit de la force de gravitation appliquée aux joueur et personnages dans le jeu ;
- l'apparence du joueur ;
- les écrans (images et textes) qui s'affichent en cas de victoire ou de défaite ;
- les sons joués dans le jeu.

L'utilisateur va pouvoir agrémenter sa scène à l'aide de différents éléments.

Tous les éléments codables de la scène possèdent leur propre page de script qui gouverne leur comportement et qui peut être modifiée. Le paysage et les plateformes ne sont pas codables.

Joueur

L'objet joueur est l'avatar que le joueur humain pourra contrôler sur tablette ou sur ordinateur, en le faisant se déplacer, sauter et tirer.

Le joueur est un élément codable accessible depuis l'ensemble de la scène, car il est unique ; il n'existe qu'un seul joueur dans la scène alors qu'il peut exister plusieurs personnages ou bonus.

Que ce soit depuis sa propre page de script ou depuis les pages de script des bonus ou personnages, on peut toujours accéder au joueur et à ce que l'on appelle ses propriétés (sa vie, son score, la hauteur de son saut, sa vitesse, etc).
À ce titre, on peut donc utiliser le code du joueur pour héberger des variables (cf. : chapitre Variables) auxquelles les autres éléments de la scène pourront accéder.

Personnages Non Joueur (PNJ)

Les PNJ sont des éléments codables de la scène. Il s'agit de personnages que l'on ne peut pas contrôler directement : si on veut leur affecter des comportements, il va falloir les coder.

Par exemple, si on veut déclencher un comportement lorsque le joueur approche, il faut tester la distance qui sépare le PNJ du joueur.

Plateformes

Les plateformes sont des éléments non codables de la scène, que le personnage ne peut pas traverser, mais sur lesquelles il peut marcher.

Paysages

Le paysage est l'image de fond qui habille le niveau. Il ne peut y avoir qu'une seule image et elle n'est pas codable.

Bonus

Les bonus sont des éléments codables de la scène. À la différence du joueur et des PNJ ont ne peut les déplacer. Par défaut, ils se comportent comme des plateformes : on peut marcher dessus. Pour permettre au joueur de passer au travers, il est nécessaire de modifier la valeur de la propriété « traversable ».

Entités

Le bloc « entité » permet de spécifier quel élément de la scène va exécuter une action. Dans GameCode, il existe 3 entités différentes :

Bloc	Script
	personnage objet joueur

On distingue ces trois entités car elles diffèrent par leurs actions et propriétés. Par exemple :

- un objet n'a pas de « vie » ;
- le joueur a déjà des contrôles par défaut, on ne peut donc pas utiliser de fonction pour le faire avancer.

Actions

On peut associer une « action » à une « entité ».

Exemples :

Bloc	Script

	joueur.dit('Hello');
	
	personnage.avance();
	
	objet.disparait();

Propriétés

Au début du jeu, vous devez définir les caractéristiques de votre joueur et de chacun de vos personnages.

Exemples :

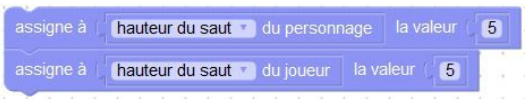
- Nous définissons le nombre de vies à 3.

Bloc	Script

	personnage.vie = 3;
	
	joueur.vie = 3;

- Nous définissons la hauteur de saut à 5, c'est-à-dire que le personnage ou le joueur sautera à une hauteur équivalente à 5 cases.

Bloc	Script

	personnage.hauteurSaut = 5;
	
	joueur.hauteurSaut = 5;

- Nous définissons la vitesse de déplacement à 15, c'est-à-dire que le personnage ou le joueur se déplacera à une vitesse équivalente à 15 cases. Plus la vitesse est grande, plus le personnage ou le joueur ira vite.

Bloc	Script

	personnage.vitesse = 15;
	
	joueur.vitesse = 15;

- Nous définissons la cadence de tir à 2, c'est-à-dire qu'une durée de 2 secondes s'écoulera entre 2 balles tirées par un personnage ou le joueur. Plus la cadence de tir est petite, plus la fréquence des balles est rapprochée.

Bloc	Script

	personnage.cadenceDeTir = 5;
	
	joueur.cadenceDeTir = 5;

Variables

Une variable est un contenant, dans lequel on peut stocker une valeur. La valeur contenue dans une variable peut changer au cours de l'exécution du programme.

Dans GameCode, on distingue deux types de variables : les variables de script et les propriétés d'entités.

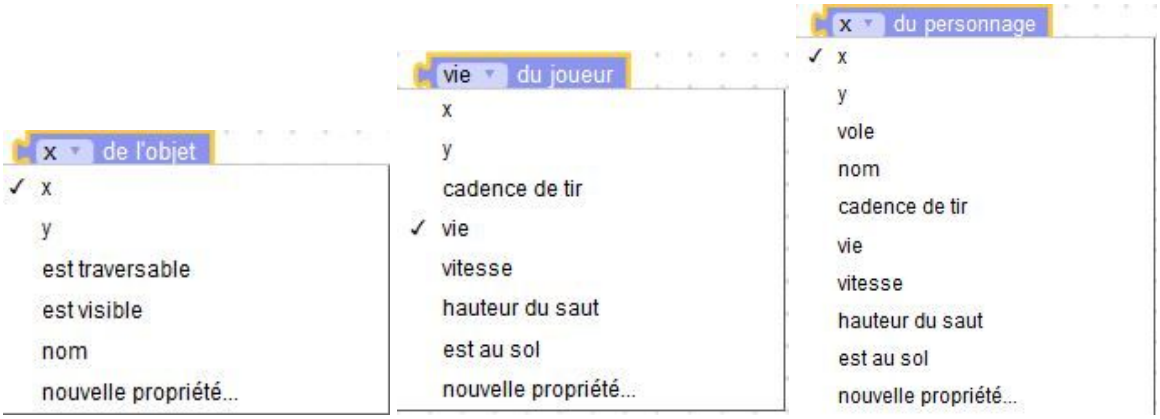
Variables de script

Les variables de script sont des variables auxquelles on ne peut faire appel que localement, au sein de la page de script de l'élément (joueur, personnage ou bonus) où elles sont déclarées.

Propriétés

Les propriétés des éléments sont les variables propres à chaque entité.

Le joueur, les personnages et les bonus ont déjà un certain nombre de variables définies qui sont consultables en dépliant la liste de ces blocs :



Ces propriétés ont un impact sur la scène et le gameplay car elles définissent des comportements et valeurs de base.
Elles peuvent être modifiées individuellement pour créer des comportements différents.

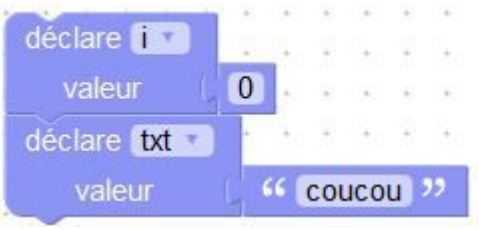
Exemples :

Bloc	Script	Commentaire
vie du joueur	joueur.vie	Permet de consulter / modifier le nombre de points de vie du joueur
vole du personnage	personnage.vole	Permet de consulter / paramétrer la capacité de voler du personnage
est visible de l'objet	objet.estVisible	Permet de rendre visible ou invisible un objet

ASTUCE : Dans GameCode le joueur est un élément accessible depuis tous les autres objets ou personnages, ses propriétés sont donc accessibles par tous. Si on lui attribue une nouvelle propriété, celle-ci sera accessible partout.

Déclarer une variable

En javascript, on déclare une variable à l'aide du mot-clé **var**, et on peut directement lui attribuer une valeur à l'aide du signe **=**.

Bloc	Script
	var i = 0; /*la variable "i" a pour valeur un entier */ var txt = 'coucou'; /* la variable "txt" a pour valeur une chaîne de caractères */

Modifier une variable

Pour modifier la valeur contenue dans une variable on a besoin du bloc « assigner à ». En javascript on utilise le signe « **=** ».

Bloc	Script
	=

	var hauteur = 0;
	
	hauteur = personnage.y - joueur.y;

Si on veut incrémenter (augmenter) ou décrémenter (diminuer) la valeur de cette variable, on a besoin de la valeur contenue par la variable, à laquelle on va ajouter ou enlever un certain nombre d'unités.

Bloc	Script
	monChiffre = monChiffre - 1;
	joueur.vie = joueur.vie - 1; /* la variable "txt" a pour valeur une chaîne de caractères */

Ici, si le joueur avait 6 points de vies, sa vie est maintenant de la valeur « 6 - 1 ».

Si on avait fait :

Bloc	Script
	monChiffre = - 1;
	joueur.vie = - 1;

La valeur de la vie du joueur serait à « -1 ».

On peut aussi utiliser une variable pour simplement accéder à sa valeur, sans la modifier.

Exemple :

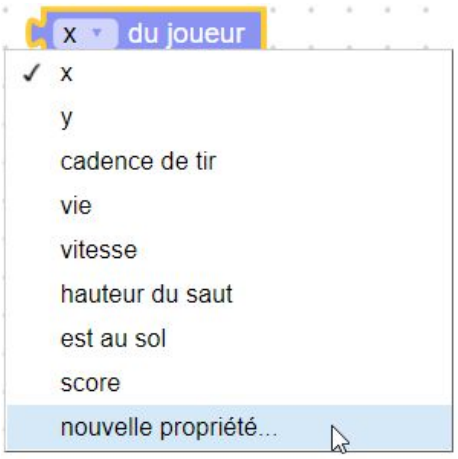
Bloc	Script

	if (personnage.vie == 0)
	{
	victoire();
	}

Création d'une nouvelle propriété

On peut aussi ajouter une propriété à une entité.

Bloc	Script

	joueur.score = 0;
	
	personnage.dureeDeVie = 0;
	
	objet.interrupteur = vrai;

Tous les objets de ce type ont ensuite cette nouvelle propriété, et elle se manipule de la même façon que toutes les autres variables.

Exemple :

Si on veut compter toutes les fleurs que le joueur récupère dans un score :

- Dans le script du joueur, on va commencer par créer une nouvelle propriété du joueur, qu'on appellera score, et à laquelle on donnera une valeur de départ.

Bloc	Script
	joueur.score = 0;

On appelle le fait de donner une valeur de démarrage à une variable « **initialiser une variable** », c'est-à-dire lui donner une valeur initiale.

- Dans le script de chaque fleur, on va pouvoir accéder à cette nouvelle variable score, et la modifier, au moment de la collision avec le joueur (quand le joueur arrive) :

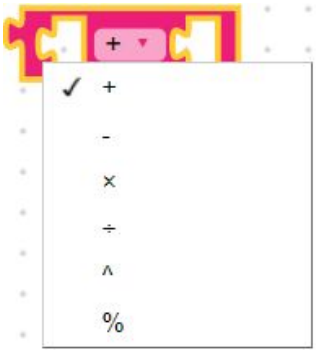
Bloc	Script

	quand('joueur arrive', fonction ()
	{
	joueur.score = joueur.score + 1;
	})

Opérations

En code, nous trouvons les opérateurs suivants :

- « + » : addition
- « - » : soustraction
- « * » : multiplication
- « / » : division
- « ^ » : exposant. Il permet d'élever un nombre à une puissance
- « % » : « modulo ». Il permet d'obtenir le reste d'une division entière



Exemples :

3 / 2 = 1,5
6^3 = 6*6*6 = 216
17%7 = 3, car 17 divisé par 7 donne 2 et il reste 3
15%5 = 0, car 15 divisé par 5 donne 3 et il reste 0

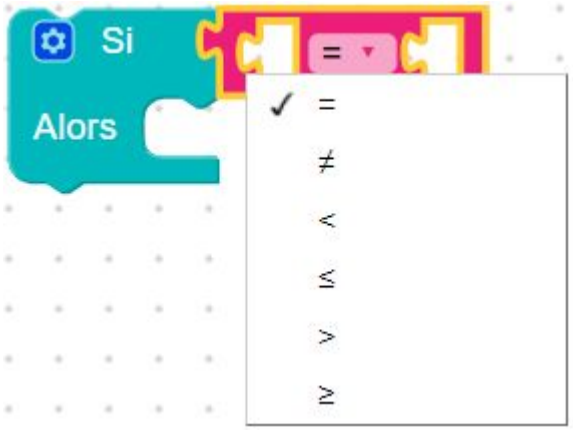
Pour modifier une variable par le calcul, il existe quelques raccourcis :

Exemples :

i = i + 1 peut également s'écrire i += 1 ou encore i++
i = i + 10 peut également s'écrire i += 10

Comparateurs

Pour comparer des valeurs, nous utilisons les signes mathématiques suivants :

Bloc	Script
	== /* renvoie vrai si les deux éléments comparés sont égaux */ != /* renvoie vrai si les deux éléments comparés sont différents */ < /* renvoie vrai si la valeur de l'élément de gauche est strictement inférieur à celle de l'élément de droite */ <= /* renvoie vrai si la valeur de l'élément de gauche est inférieur ou égale à celle de l'élément de droite */ > /* renvoie vrai si la valeur de l'élément de gauche est strictement supérieur à celle de l'élément de droite */ >= /* renvoie vrai si la valeur de l'élément de gauche est supérieur ou égale à celle de l'élément de droite */

En javascript on peut aussi observer des comparaisons utilisant le signe « **===** ».
Ce test compare les valeurs de deux éléments mais aussi leur type.

Exemple :

```
(i === 0) /*Cette écriture compare les valeurs et aussi le type*/
```

En programmation le type permet de différencier les variables, par exemple **var test = 0** et **var test = "0"** n'ont pas le même type, la première est un chiffre, la deuxième est un texte, ou chaîne de caractères.

Exemple :

On compare le caractère 'A' à la valeur 65. En ASCII, 'A' vaut 65 donc la comparaison 'A' == 65 est vraie.

```
'A' === 'A' /*Vrai*/
'A' === 'B' /*Faux*/
```

```
'A' === 65 /*Faux*/
'A' == 65 /*Vrai*/
```

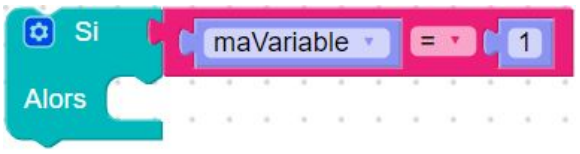
Conditions

Une condition permet de définir une tâche à effectuer lorsqu'une hypothèse est vérifiée.

À l'aide des comparateurs, les conditions permettent d'écrire du code, qui sera mis dans un bloc SI/ALORS. SI la condition située après le terme « si » est vraie, ALORS le code contenu dans la partie « alors » du bloc sera exécuté. En javascript, cette partie correspond à celle encadrée par les accolades « **{ }** ».

Exemple 1 :

Bloc	Script

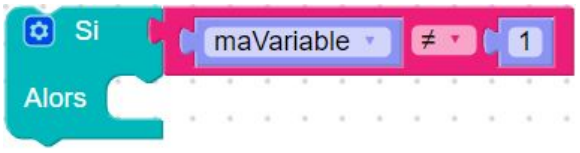
	si (maVariable == 1)
	{
	
	}

Cette condition évalue l'égalité entre « maVariable » et la valeur 1. Il y a deux cas de figure :

- « maVariable » vaut 1. Cette comparaison renvoie la valeur vrai (true). Le code contenu après ALORS est exécuté.
- « maVariable » ne vaut pas 1. Cette comparaison renvoie la valeur faux (false). Le reste du bloc SI est ignoré et le code contenu après ALORS n'est pas exécuté.

Exemple 2 :

Bloc	Script

	si (maVariable != 1)
	{
	
	}

Cette condition évalue l'inégalité entre « maVariable » et la valeur 1. Il y a deux cas de figure :

- « maVariable » ne vaut pas 1. Cette comparaison est vraie, « maVariable » est différente de 1. La comparaison renvoie la valeur vrai (true). Le code contenu après ALORS est exécuté.

- « maVariable » vaut 1. Cette comparaison renvoie la valeur faux (false). Le reste du bloc SI est ignoré et le code contenu après ALORS n'est pas exécuté.

Autres exemples :

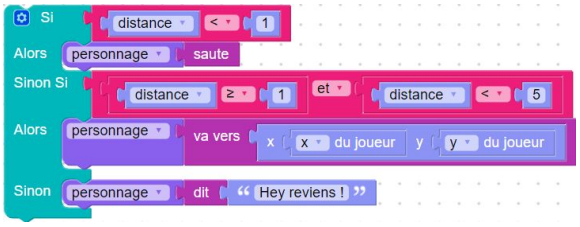
Bloc	Script

	si (joueur.vie <= 1)
	{
	joueur.dit('au secours !');
	}

Si l'expression « joueur.vie <= 1 » n'est pas vérifiée par la condition, c'est que la vie du joueur est strictement supérieure à la valeur 1. Le programme n'exécutera donc pas le code inscrit entre les accolades {}.

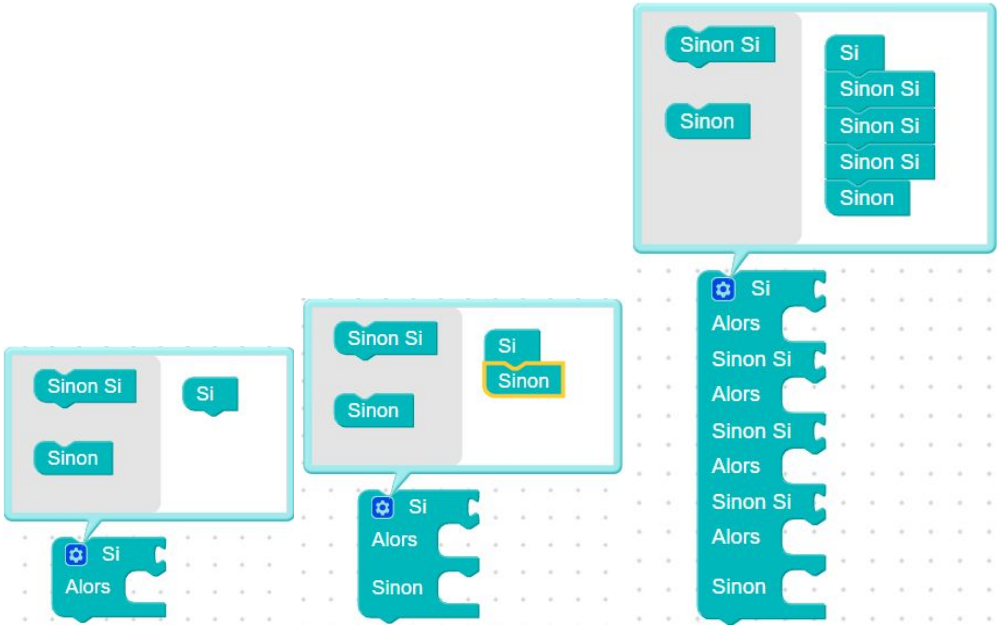
On peut imaginer des conditions plus complexes :

Bloc	Script

	si (distance < 1)
	{
	personnage.saute();
	}
	sinon si (distance >= 1 && distance < 5)
	{
	personnage.vaVers(joueur.x, joueur.y);
	}
	sinon
	{
	personnage.dit('Hey reviens !');
	}

Comme indiqué dans l'exemple, il est possible d'utiliser des comparateurs et règles logiques dans les conditions.

Dans Blockly, pour modifier la condition, on utilise la roue dentée présente en haut à gauche du bloc « si » :



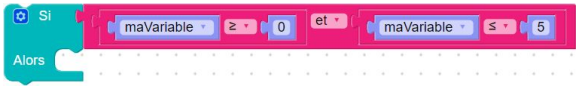
Règles logiques

Les symboles logiques ET et OU permettent de donner un caractère inclusif ou exclusif à une comparaison.

Bloc	Script
	&& /* et */ || /* ou */

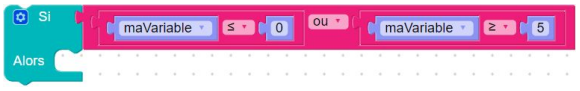
Exemples :

Bloc	Script

	si ((maVariable >= 0) && (maVariable <= 5))
	{
	
	}

Cette condition est vérifiée si la valeur de « maVariable » est à la fois supérieure ou égale à 0, et inférieure ou égale à 5. En clair, cette condition est vraie si la valeur de « maVariable » est comprise entre 0 et 5.

Bloc	Script

	si ((maVariable <= 0) || (maVariable >= 5))
	{
	
	}

Les deux éléments de cette vérification sont maintenant traités séparéments et le résultat sera vrai si la valeur de « maVariable » est inférieure ou égale à 0 OU si la valeur de « maVariable » est supérieure ou égale à 5.

Si jamais on se trompe et que l'on met ET, le code présent dans la condition ne sera jamais exécuté, car il faudrait que « maVariable » soit à la fois plus grande que 5 et plus petite que 0. Il faut donc bien faire attention aux conditions que l'on utilise, et au choix de mettre ET et OU.

Les boucles

La boucle toujours

Dans GameCode la boucle « toujours » fait s'exécuter le code qu'elle contient à chaque frame (image).

Dans ce premier exemple, la boucle « toujours » ajoute 1 point de score au joueur à chaque frame. Le score augmentera donc de manière continue dès le lancement du jeu.

Bloc	Script

	toujours(fonction ()
	{
	joueur.score = joueur.score + 1;
	})

Dans ce deuxième exemple, la boucle « toujours » essaie de faire tirer le personnage à chaque frame. En réalité, le personnage a une cadence de tir maximale qui est définie. Dans ce cas, la boucle « toujours » fait tirer le personnage à sa cadence maximale, et non pas à chaque frame.

Bloc	Script

	toujours(fonction ()
	{
	personnage.tire();
	})

Attention, si on met une animation dans une boucle « toujours », comme « danse», la boucle relance l'action indéfiniment, depuis la première frame de l'animation. L'animation « danse» est composée de plusieurs frames et seule la première aura le temps d'être exécutée par la boucle « toujours », laissant l'impression d'une image fixe.

Bloc	Script

	toujours(fonction ()
	{
	personnage.danse();
	})

En programmation, le code se lit ligne par ligne, de haut en bas. En général, une boucle empêche le programme de passer aux lignes de code suivantes, tant qu'elle n'est pas finie. Ici, les boucles sont des fonctions qui permettent de répéter une action, sans pour autant bloquer l'exécution du programme.

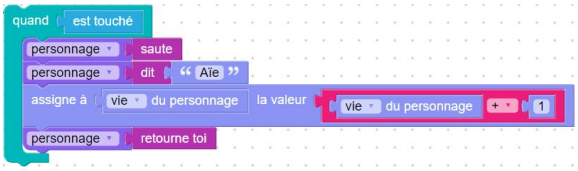
Événement

La programmation d'événements permet de déclencher l'exécution d'un bout de code uniquement lorsque l'événement se produit.

Ce code ne s'exécute pas tant que le déclencheur n'est pas atteint. Une fois que ce dernier est atteint, les actions définies dans l'événement se réalisent.

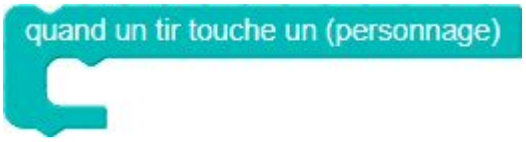
Dans cet exemple, on code la réaction d'un personnage quand il est touché par un projectile.

Bloc	Script

	quand ('est touché', fonction ()
	{
	personnage.saute();
	personnage.dit('Aïe');
	personnage.vie -= 1;
	personnage.retourneToi();
	})

Les différents événements sous GameCode

Quand un tir touche un personnage

Bloc	Script
	quand ('tir touche', fonction (personnage) { })

Ce bloc ne peut être placé **que dans le code du joueur**. Il permet de connaître le personnage touché, dans le but de lui faire exécuter du code.

Dans ce cas précis, même si l'on se situe dans la page de script du joueur, on peut affecter un comportement au personnage touché, car on reçoit sa référence (en javascript, « personnage » est passé en paramètre de la fonction). On distingue ainsi quel personnage en particulier, parmi tous ceux de la scène, a été touché.

Si on utilise cet événement, tous les personnages touchés par un tir du joueur effectueront systématiquement la même action.

Exemple :

Ici, on va faire sauter tous les personnages qui viennent d'être touchés.

Bloc	Script

	quand ('tir touche', fonction (personnage)
	{
	personnage.saute();
	})

Quand

Ce bloc se combine à un « message déclencheur » pour construire un événement. Lorsque le message est propagé par le programme, le code associé au bloc **Quand** est exécuté.

Exemple :

Bloc	Script
	quand ('est bloqué', fonction () { personnage.retourneToi(); })

Quand un personnage arrive

Bloc	Script
	quand ('personnage arrive', fonction () { })

Présent dans la page de script du joueur, d'un personnage ou d'un bonus, ce bloc permet de détecter la collision avec un personnage.

- « Arrive » correspond à l'instant où le personnage rentre en contact avec l'élément associé à ce code (le joueur, un autre personnage ou un bonus).
- « Est devant » correspond au moment où le personnage est superposé à l'élément associé à ce code. Cet événement est continu et se déclenche à chaque frame, tant que les objets concernés sont en contact.
- « Part » correspond à l'instant où le personnage quitte l'élément associé à ce code. Cet événement est ponctuel et se déclenche une seule fois, quand le contact est rompu.

Exemple :

Bloc	Script

	quand ('personnage arrive', fonction ()
	{
	personnage.dit('Au revoir');
	})

Remarque :

Il est possible de détecter si le joueur « arrive », « est devant » ou « part » d'un objet ou d'un personnage avec les blocs suivants :

Bloc	Script
	quand ('joueur arrive', fonction () { }) 'joueur est devant' 'joueur part'

Fonctions

Une fonction permet de rassembler un ensemble d'instructions.
Sous GameCode, il n'est possible de créer des fonctions qu'en langage javascript, pas en bloc.

Exemple :

```
fonction devenirPlusFort()
{
    joueur.vie = joueur.vie + 10;
    joueur.vitesse = joueur.vitesse + 5;
    joueur.cadenceDeTir = 0,1;
}
```

Pour appeler cette fonction il suffit d'écrire :

```
devenirPlusFort();
```

Exemples d'utilisation :

Sans l'usage des fonctions	Avec les fonctions

si (joueur.vie == 1)
{
joueur.vie = joueur.vie + 10;
joueur.vitesse = joueur.vitesse + 5;
joueur.cadenceDeTir = 0,1;
}
	si (joueur.vie == 1)
	{
	devenirPlusFort();
	}

quand ('est touché', fonction()
{
joueur.dit('Aïe');
joueur.vie = joueur.vie - 1;
})
	fonction direAie()
	{
	joueur.dit('Aïe');
	joueur.vie = joueur.vie - 1;
	}
	
	quand ('est touché', direAie());

quand ('tir touche', fonction(personnage)
{
joueur.dit('Pardon');
joueur.score = joueur.score - 1;
})
	fonction direPardon()
	{
	joueur.dit('Pardon');
	joueur.score = joueur.score - 1;
	}
	
	quand ('tir touche', direPardon());

Les références sur éléments

En script, il est possible d'attribuer une référence à un personnage ou un bonus dans le but de pouvoir ensuite l'identifier et le distinguer des autres personnages et bonus.

Pour cela il faut commencer par donner un nom à notre bonus ou personnage.

Exemple :

```
objet.nom = "nomObjet";
personnage.nom = "nomPersonnage";
```

Ensuite, pour cibler ce bonus ou personnage, depuis la page de script du joueur (ou n'importe quel autre élément) on utilise le mot-clé **def** pour définir une nouvelle variable, dans laquelle on stocke la référence de l'élément ciblé grâce aux fonctions *trouverObjet()* et *trouverPersonnage()*.

Exemple :

```
def monObjet = trouverObjet("nomObjet");
def monPerso = trouverPersonnage("nomPersonnage");
```

Une fois fait, on peut accéder aux méthodes et propriétés de l'élément en utilisant la variable définie.

Exemple :

```
def monPerso = trouverPersonnage("nomPersonnage");
monPerso.danse();
```

On peut aussi stocker la référence de l'élément dans une propriété du joueur. Ainsi l'élément sera accessible n'importe où puisque les propriétés du joueur sont accessibles à tout moment.

Exemple :

```
joueur.monPerso = trouverPersonnage("nomPersonnage");
joueur.monPerso.danse();
```

Attention :
Il est possible de récupérer une référence nulle si le bonus ou le personnage correspondant n'a pas été trouvé. Cela arrive si le code *trouverObjet()* ou *trouverPersonnage()* est exécuté avant même que l'élément ne soit nommé (exemple : si le script contenant trouverPersonnage() est exécuté avant le script personnage.nom = "nomPersonnage";).
Pour éviter cela, il est conseillé d'utiliser les fonctions *trouverObjet()* ou *trouverPersonnage()* dans un événement.

Exemple :

Page de script du personnage :

```
personnage.nom = "toto";
```

Page de script du joueur :

```
Quand("personnage arrive", fonction()
{
        def toto = trouverPersonnage("toto");
});
```

Ici, le code du personnage est exécuté dès le lancement du programme, alors que la fonction *trouverPersonnage()* sera exécutée plus tard, lorsque l'événement sera déclenché.

Il est aussi possible de récupérer cette référence dès le début, si besoin, et de se passer d'un événement. Dans ce cas on va utiliser une boucle pour exécuter les fonctions *trouverObjet()* ou *trouverPersonnage()* jusqu'à temps qu'on obtienne la référence de l'élément.

Exemple :

Page de script du personnage :

```
personnage.nom = "toto";
```

Page de script du joueur :

```
/* On définit une variable avec une valeur nulle par défaut */
def toto = null;

/* Tant qu'on a pas récupéré la référence du personnage on relance la boucle */
while (toto == null)
{
    /* La fonction trouverPersonnage() renvoie la référence du personnage si elle la trouve, la valeur "null" sinon */
    toto = trouverPersonnage("toto");
}

/* On est sortie de la boucle. On a donc l'assurance d'avoir la référence du personnage. On peut donc l'utiliser */
toto.danse();
```

Les erreurs fréquentes

Confondre joueur et personnage

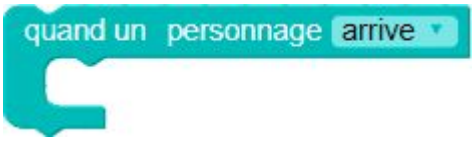
Le « joueur » est l'avatar  que vous contrôlez en jouant le jeu.

Le « personnage »  est le Personnage Non Joueur (PNJ) que vous placez dans la scène. Il peut y avoir plusieurs personnages dans la scène.

On ne peut pas utiliser le bloc 

sur un personnage pour lui faire effectuer une action.

Exemple 1 :



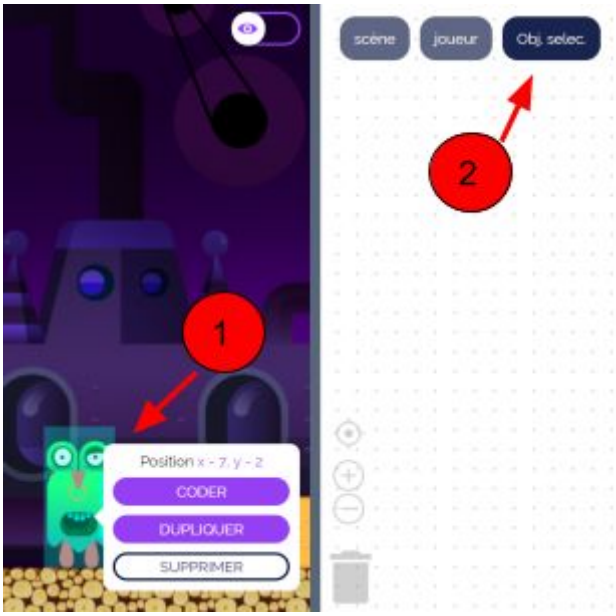
Ce bloc permet de détecter si un personnage touche un joueur, personnage ou objet.
Il ne peut pas servir à détecter si le joueur touche un personnage.

Exemple 2 :

Ce code ne fonctionnera pas sur le joueur.

Ne pas coder le bon élément de jeu

Attention à toujours bien sélectionner l'élément (joueur, PNJ, bonus) dont vous souhaitez coder le comportement avant de passer à l'onglet de code.



Exemple :

sur le joueur ne fonctionnera pas.

Confondre actions et événements

Attention : Les blocs actions sont de couleur mauve tandis que les messages d'événements sont de couleur bleu.
Certains messages et actions se ressemblent, cependant ils ne s'utilisent pas de la même façon.

Exemple :

Ce code ne fonctionnera pas. Dans cet exemple, « Disparaît » est un message d'événement et non une action (

).

Modifier le code de l'élément "scène"

Dans la zone de code on peut trouver cet élément :

Le code associé correspond à la valeur de la force de gravitation s'exerçant sur le niveau. Il est possible de la modifier.

Important :
Tout autre code ajouté à cet emplacement ne sera pas interprété par GameCode et sera même source d'erreur.
C'est le premier endroit à vérifier en cas de création non fonctionnelle.

Blocs incomplets

Le fait de laisser des blocs incomplets peut générer une création dysfonctionnelle. Il est recommandé de dézoomer chaque page de script pour identifier si des blocs isolés ou incomplets gênent le bon fonctionnement du programme.