

Event Extender

Version 4

nom complet : 4.5.6

Lien de téléchargement :

[Lien de téléchargement](#)

Table des matières

Prélude.....	3
<i>Remerciements</i>	3
Les variables et interrupteurs, interface d'accès rapide.....	4
Variables et interrupteurs locaux, interface d'accès rapide.....	6
Déclencheurs personnalisés.....	8
Découpe des nombres (isoler chaque chiffre).....	9
Le module Command.....	9
<i>Comment utiliser les commandes ?</i>	9
<i>Commandes standards</i>	11
<i>Commandes systèmes</i>	15
<i>Commandes d'opérandes sur les objets</i>	16
<i>Commandes de partie</i>	17
<i>Commandes de lecture de données des monstres</i>	20
<i>Commandes de manipulation de la souris</i>	22
<i>Commandes de manipulation du clavier</i>	26
<i>Commandes de manipulation des évènements</i>	32
<i>Commandes de manipulation des images</i>	37
<i>Multi-Panoramas</i>	45
<i>Module de la date et de l'heure</i>	47
<i>Gestion des sauvegardes</i>	48
<i>Gestion des zones</i>	49
<i>Zone de texte saisissable au clavier</i>	52
<i>Interface réseau (Sockets)</i>	57
<i>Commandes diverses</i>	59
<i>Conclusion des commandes</i>	60
Les outils complémentaires	61
<i>Le testeur de teintes</i>	61
<i>Le testeur de script</i>	62
<i>La base de données alternative</i>	64
L'éditeur de commandes.....	67
<i>Mise en garde</i>	68
<i>Téléchargement</i>	68
<i>Installation</i>	68
<i>Liste des commandes</i>	68
<i>Créer une commande particulière</i>	69
Conclusion finale.....	70
<i>Conditions d'utilisations</i>	70

Prélude

L'objectif de ce script est d'offrir une multitude de commandes complémentaires pour la création de systèmes via l'event-making, qui est, selon moi, une pratique ingrate et compliquée.

C'est pour ça que j'ai écrit ce script, dans la même veine que l'Event-language (devenu GeeX Make) de Roys et Avygeil.

L'event extender est une collection d'outils faciles à utiliser qui permettent de s'affranchir de certaines règles et de certaines obligations dans l'event-making en règle générale.

De plus, étant principalement basé sur des appels de scripts, chaque argument peut être substitué par une variable, chose impossible avec l'interface actuelle de RPG Maker VX Ace.

L'objectif de ce document est d'offrir une base de travail. Soit une documentation. L'Event Extender est un script qui a été prévu pour être étendu, donc n'importe qui peut suggérer/implémenter sa propre commande (la notion de commande sera expliquée plus tard dans ce même document).

Ce document peut paraître un peu abrupt à lire mais une fois que les différents concepts du script sont appréhendés, je vous assure qu'il est possible de gagner énormément de temps dans la création de vos systèmes.

De plus, nous verrons, plus tard, que les « apprentis scripteurs » pourront se servir de plusieurs composantes de l'Event Extender pour la création de script, de manière transparente.

Remerciements

L'Event Extender est un gros projet, je n'aurai jamais pu le terminer sans l'aide inestimable de Zeus81 et de Nuki que je remercie tout particulièrement pour leurs conseils, leurs solutions. Et aussi parce qu'ils ont tous les deux été une source d'inspiration (comme beaucoup d'autres). Je remercie aussi chaleureusement Magi, Hiino, XHTMLBoy, Zangther, Lidenvic, Raho, Joke, Al Rind, Testament, Heos, S4suk3, Avygeil, Tonyryu, Siegfried, Berka, Nagato Yuki, Fabien, Roys, Raymo, Ypsoriama, Amalrich Von Monesser, Uli, ParadoxII, Loly74, 2cri, Kmkzy pour leurs conseils techniques comme moraux ainsi que pour des suggestions de commandes, de concepts.

J'espère n'avoir oublié personne et si c'est le cas, je vous présente mes plus sincères excuses !

L'origine du script est de Nuki, auteur des variables locales.

Zeus81 aura été d'une titanesque aide pour la réalisation de plein de commandes (la gestion clavier/souris, les variables locales).

La commande Buzz a été initialement créée par Fabien.

Je remercie aussi chaleureusement Uli pour sa bienveillante relecture (qui m'a beaucoup aidé pour la reformulation de phrases douteuses ainsi que sa relecture orthographique). Je vous (et lui) présente d'avance des excuses pour avoir ajouté mon lot d'erreurs dans cette nouvelle version de la documentation.

Les variables et interrupteurs, interface d'accès rapide

Les variables et les interrupteurs sont très utilisés dans la création de jeu RPG Maker. En général, les fenêtres du logiciel permettent d'affecter des variables à certaines valeurs (par exemple la position x et y d'une image ou encore les coordonnées de téléportation du héros), cependant, leur utilisation dans un appel de script est long. Pour appeler une variable et un interrupteur, la syntaxe est comme ceci :

- *\$game_variables[id désiré]*
- *\$game_switches[id désiré]*

Le premier objectif de ce script aura été de simplifier l'accès aux variables et aux interrupteurs. Maintenant il suffit d'utiliser les lettres V (pour les variables) et S (pour les interrupteurs).

Leur syntaxe est donc comme ceci :

- *V[id de la variable]*
- *S[id de l'interrupteur]*

L'attribution de valeur à une variable devient donc très facile. Par exemple, pour donner la valeur 134 à la variable 98, il suffira de faire, dans un appel de script (ou dans un script complet) :

V[98] = 134

Étant donné qu'il est possible d'utiliser cette syntaxe partout, il serait tout à fait envisageable de faire :

*V[V[12]] = V[1]+V[2]*V[3]*

Ce qui correspond à dire que la valeur de la variable 12 sera l'ID de la variable que l'on modifie, et qu'on lui donnera la valeur de la variable 1 plus le produit de la variable 2 et de la variable 3.

Il existe une série d'opérateur en plus de + et *, voici une liste rapide (et potentiellement non exhaustive) des opérateurs possibles :

- *+ addition*
- *- soustraction*
- ** produit*
- */ division (attention aux divisions entières, 5/2 = 2)*
- *** puissance (3 ** 2 = 9)*
- *% Modulo (reste de la division euclidienne)*

Il existe des fonctions (et des méthodes) qui permettent la manipulation des nombres (par exemple, « log », « sin », « cos ») cependant, nous n'allons pas nous attarder sur ce sujet. Si un lecteur désire aller plus loin avec les nombres, il peut se référer à la documentation de Ruby (que je conseille grandement comme lecture).

Il est aussi possible de compresser certains opérateurs. Par exemple, imaginons que je veuille ajouter un nombre à une variable ? La solution triviale serait :

$V[12] = V[12] + 1$

Cependant, il existe, pour les opérateurs $+$ $-$ $/$ $*$ et $\%$ une syntaxe allégée. Par exemple :

- $V[1] += 2$ ajoutera 2 à la variable 1
- $V[2] -= 3$ soustraira 3 à la variable 2
- $V[3] *= V[1]$ la variable 3 sera multipliée par la variable 1
- $V[4] /= 2$ divisera la variable 4 par 2
- $V[5]\% = 2$ donnera le reste de la division entière/euclidienne de la valeur de la variable 5 par deux.

Il est aussi possible de limiter à une ligne l'assignation de plusieurs variables, par exemple :

- $V[1], V[2], V[3] = 8, 9, 10$
- $V[1] = V[2] = V[3] = 7$
- $V[1..19] = 10$

Dans le premier exemple, la variable 1 vaudra 8, la 2, 9 et la 3, 10.

Dans le second exemple, les 3 variables vaudront 7.

Dans le troisième exemple la plage de variables de 1 à 19 (compris) vaudra 10.

Il existe d'autre raccourcis syntaxique (par exemple, intervertir rapidement deux variables : $V[1], V[2] = V[2], V[1]$) mais je conseille tout de même une formation plus poussée en Ruby pour aller plus loin dans l'expressivité de ces fonctions.

Comme pour les variables, il est possible d'accéder rapidement aux interrupteurs via $S[id]$, un interrupteur ne peut admettre que deux valeurs, *true* ou *false*. La valeur *true* correspondant à l'interrupteur « activé » et la valeur *false* à l'interrupteur désactivé. Comme pour les variables, les raccourcis d'assignation fonctionnent aussi, $S[1], S[2] = S[2], S[1]$ etc.

Les interrupteurs possèdent aussi leurs opérateurs, on appelle ça des opérateurs logiques. Une fois de plus, dans cette documentation, nous ne nous étalerons pas sur la notion de logique mais retenir ces deux opérateurs logiques.

- L'opérateur « *and* », renvoie *true* si les deux membres sont *true*. Renvoie *false* sinon.
- L'opérateur « *or* », renvoie *true* si un des deux membres est *true*, si aucun des deux ne l'est, renvoie *false*.

Plus tard, lorsque nous étudierons les commandes, nous verrons que certaines commandes renvoient des « booléens », booléen étant le nom qu'on donne à *true* et *false*.

Une condition (en event-making comme programmation) ne prend qu'un booléen. Si le booléen est vrai on rentre dans la condition, s'il est faux, on rentre dans le « sinon ».

Outre les interrupteurs, il est possible de « créer » des booléens via des questions. Par exemple :

- $x == y$
si x est égal à y, cette expression vaudra « true » (« false » sinon)
- $x != y$
si x n'est pas égal à y, cette expression vaudra « true » (« false » sinon)
- $x > y$
si x est plus grand que y, cette expression vaudra « true » (« false » sinon)
- $x < y$
si x est plus petit que y, cette expression vaudra « true » (« false » sinon)
- $x >= y$
si x est plus grand ou égal à y, cette expression vaudra « true » (« false » sinon)
- $x <= y$
si x est plus petit ou égal à y, cette expression vaudra « true » (« false » sinon)

Il est aussi possible d'inverser un booléen, en lui mettant un point d'exclamation devant, par exemple, $!(true)$ sera égal à false et $!(9 >= 10)$ vaudra true.

Il existe d'autres opérateurs cependant, l'objectif de cette page n'est pas d'apprendre la programmation et donc je vous conseillerai cette page qui résume très bien le fonctionnement des conditions logiques :

<http://fr.wikiversity.org/wiki/Ruby/Conditions>

Ils utilisent $\&\&$ et $\|\|$ ce qui équivaut respectivement à and et or.

Retenez pour le moment qu'un interrupteur ne peut avoir que deux valeurs possibles, true ou false.

Je conclus cette partie sur l'interface d'accès rapide aux variables et interrupteurs en proposant un lien vers un tutoriel rédigé par Nuki qui explique son fonctionnement plus en détail :

[Lien vers son tutoriel](#)

Variables et interrupteurs locaux, interface d'accès rapide

Cette partie peut paraître plus complexe mais reste relativement simple dans le fond. Nous allons aborder le concept de variables locales.

Concrètement, une variable locale est une variable qui peut contenir exactement la même chose qu'une variable globale (variable normale de RPG Maker), sauf qu'elle est référencée par une adresse composée de l'ID de la map, de l'ID de l'événement et de son ID à elle.

L'avantage est qu'il est possible de créer une variable en ne spécifiant que son

ID à elle, et le script automatisera l'attribution des deux autres ID.

Quel est l'avantage ? Cela évite la création de variables globales pour des utilisations uniquement au sein d'un événement, ça permet de copier/coller ses événements (qui utilisent des variables locales) sans devoir modifier les variables qu'ils utilisent et ça augmente aussi considérablement le nombre de variables disponibles.

Personnellement, je m'en sers souvent pour stocker des résultats dont je n'ai besoin QUE dans un événement ou qui sont propres à cet événement.

De plus il est possible d'accéder aux variables d'un autre événement et même d'une autre map car ces adresses sont purement virtuelles, donc il est possible d'associer des variables à des événements qui n'existent pas.

Le fonctionnement des variables locales est identique à celui des variables globales, à la différence qu'il faut utiliser SV au lieu de V (SV = self variables). Dans un événement, SV[10] correspondra à SV[id_map, id_event, 10], donc il est possible de spécifier le nom complet de la variable, juste son event_id et son id, ou juste via son id. Dans le cas où l'event_id manque ou que l'event_id et le map_id manquent, le script lui attribuera l'id de map courant et l'id de l'événement courant.

Par exemple, admettons que nous ayons l'événement 4 sur la map 6 :

- SV[1] référencera la variable locale 1 de l'événement 4 de la map 6 (sera donc identique à SV[4, 1] et SV[6, 4, 1] ;)
- SV[5, 1] référencera la variable locale 1 de l'événement 5 de la map 6 (sera donc identique SV[6, 5, 1] ;)
- SV[7, 6, 1] référencera la variable locale 1 de l'événement 6 de la map 7

Tous les opérateurs étudiés dans la partie sur les variables globales fonctionnent. Voici quelques exemples. Nous allons imaginer que nos appels de scripts sont dans l'événement 8 de la map 4 :

- $SV[1] = V[2]*3$
Dans cet exemple, la variable locale référencée à 4-8-1 vaudra la valeur stockée dans la variable globale 2 multipliée par 3
- $SV[1] += 10$
On ajoute 10 à notre variable locale 1
- $SV[1], V[2] = V[2], SV[1]$
On intervertit la valeur de la variable locale 4-8-1, avec celle de la variable globale 2.
- $S[1] = SV[1] >= SV[2, 1]$
Active l'interrupteur global 1 si la variable locale 4- 8-1 est plus grande ou égale à la valeur de la variable locale 4-2-1, sinon le désactive.
- $SV[1, 2, 3] = 19$
Attribue à la variable 3 de l'événement 2 de la map 1, la valeur 19.

Les interrupteurs locaux fonctionnent comme les interrupteurs normaux avec le même système d'adressage que les variables locales. La seule différence est qu'ils sont limités à 4. Et on y accède via une lettre (A, B, C ou D) ou via un entier, 1, 2, 3 ou 4 (1 = A, 2 = B etc.) et qu'on utilise la primitive SS (SS = self switches) pour y accéder par exemple : SS["A"] (le A doit bien être entre guillemet) ou alors SS[1]. Son système d'adressage fonctionne exactement de la même manière que les variables locales donc il est possible d'accéder aux interrupteurs locaux d'autres événements.

Pour conclure, je terminerai en disant que pour ceux qui utilisaient l'ancienne version de l'Event Extender, ce changement radical de forme pour la gestion des variables locales peut paraître étrange, ceci dit, j'ai fait ce choix pour être le plus proche possible des variables globales et pour uniformiser les structures de stockage de RPG Maker. Qu'il s'agisse des variables locales ou des interrupteurs locaux, ils sont très pratiques parce qu'ils permettent de créer des événements génériques et copiables/collables sans détruire tout la logique d'un système.

Note sur l'affichage des messages

Les deux commandes événementielles d'affichage de messages (« Afficher un message » et « Afficher un texte défilant ») permettent d'afficher dans le texte, la valeur de variables au moyen de la commande \V[id], il est possible de faire pareil avec les variables locales : \SV[id] ou alors \SV[id_event, id] ou encore \SV[id_map, id_event, id].

Je vous invite à vous familiariser avec les variables locales car pour moi, c'est vraiment un outil très utile qui peut faire gagner pas mal de temps.

Déclencheurs personnalisés

L'ajout des variables locales et d'une plus grande portée aux interrupteurs locaux pose des soucis de pertinence dans les déclencheurs d'événements. En effet, les embranchements envisageables sont assez limités et ne tiennent absolument pas compte (logique me direz-vous) des apports que nous avons apportés dans les structures de stockage.

Pour pallier ce problème j'ai ajouté une fonctionnalité permettant de créer ses propres déclencheurs. Cet ajout est lié aux déclencheurs précédents ce qui veut dire qu'il est possible de les associer.

Son fonctionnement est assez simple, il suffit que la première commande d'une page soit un commentaire (dans « ajouter un commentaire » qui commence par « Trigger : » après les deux points, il faut mettre une expression booléenne. Par exemple de booléen, V[1] >= SV[3] and SV[7] == 17

exemple d'utilisation

Trigger : V[1] >= SV[3] and SV[7] == 17

déclenche l'événement si on a à la fois la valeur de la variable globale 1

supérieure à la valeur de la variable locale 3 et en même temps la valeur de la variable locale 7 égale à 17.

Ces déclencheurs personnalisés permettent de simuler la condition « Si Script » (disponible dans les commandes de condition d'évènements standards) et apporte donc plus de flexibilité dans le paramétrage de vos évènements. Il est possible d'évaluer n'importe quelle expression Ruby (cependant, attention de ne pas évaluer un élément qui n'existe pas encore !)

Découpe des nombres (isoler chaque chiffre)

L'affichage des nombres est un problème récurrent dans la création de systèmes via les évènements. Il faut les découper par chiffre et c'est souvent une opération gourmande en temps.

Comme dans sa version précédente, l'Event-Extender permet d'isoler chaque chiffre d'un nombre très facilement.

En effet il suffit de pointer le chiffre que l'on veut :

- *123654.unites donnera comme valeur 4*
- *123457.dizaines donnera 5*
- *129876.milliers donnera 9*

Il est possible d'aller jusque « centaines_millions » (les primitives disponibles sont unites, dizaines, centaines, milliers, dizaines_milliers, centaines_milliers, millions, dizaines_millions, centaines_millions).

Cette découpe s'applique à toute forme de nombre, donc il est possible de l'appliquer aux variables et aux variables locales :

V[12].unites, SV[1,2,3].milliers etc.

Le module Command

Nous allons maintenant aborder la grosse partie de cette documentation. Il s'agit du module Command.

Une commande peut être plusieurs choses. Il peut s'agir d'un outil qui donnera une valeur, d'une action.

Nous allons voir dans cette rubrique l'intégralité des commandes implémentées par défaut dans l'Event Extender (car c'est la partie la plus extensible et chaque scripteur pourra apporter sa contribution très facilement).

Comment utiliser les commandes ?

Les commandes peuvent s'utiliser de plusieurs manières. En général, quand elles donnent une valeur, elle se lie à des variables. Quand elles font une action, elles sont appelées toutes seules.

Il existe plusieurs manières d'appeler une commande. Les voici :

- *Command.nom_de_la_commande(arguments séparés par des virgules)*
- *command(:nom_de_la_commande, arguments séparés par des virgules)*

- *cmd(:nom_de_la_commande, arguments séparés par des virgules)*
- *c(:nom_de_la_commande, arguments séparés par des virgules)*

Les arguments, ce sont les données que la fonction requiert. Par exemple, la commande « percent » prend deux arguments, valeur et maximum. La commande donnera le pourcentage de valeur par rapport à maximum. Valeur et maximum sont donc les arguments de la commande :

- `percent(val, max)`
- `percent(7, 14)` (donnera 50 parce que 7 c'est 50 % de 14). 7 et 14 sont les arguments de la commande.

Avec ces manières d'appeler les commandes, ça fonctionnera aussi bien dans un appel de script que dans un script.

Il existe une syntaxe alternative uniquement pour les appels de scripts :

- *nom_de_la_commande(arguments séparé par des virgules)*
- *extender_nom_de_la_commande(arguments séparés par des virgules)*

La syntaxe préfixée de `extender` peut sembler superflue mais elle a été ajoutée dans le cas où un autre script écraserait une commande de l'Event Extender. Si une commande ne fonctionne pas comme vous l'entendez parce qu'elle a été écrasée par une autre méthode, la version préfixée de `extender_` fonctionnera (à priori). Personnellement, j'ai pris l'habitude d'utiliser `cmd(:nom_de_la_commande, arguments)` pour pouvoir utiliser mes commandes de la même manière partout, dans un script, dans un évènement etc.

Cependant, si l'idée d'emballer la commande dans sa fonction (ici, « `cmd` » en l'occurrence) vous donne de l'urticaire, sachez qu'il existe les méthodes citées plus haut.

Note pour les scripteurs, si vous voulez créer une classe ou un module qui étend naturellement les commandes, il suffit d'ajouter dans votre classe « `extend Command` » et il intégrera toutes les commandes du module.

Je vais maintenant tâcher de vous présenter toutes les commandes de la manière la plus précise possible. N'hésitez pas à les tester pendant que vous lisez cette documentation pour bien comprendre leur fonctionnement.

Certaines commandes vont vous sembler superflues (parce qu'elles existent déjà dans les commandes de RPG Maker, ceci dit, si elles ont été implémentées, c'est pour être greffable aux variables locales et appelables depuis n'importe où).

Il est très souvent courant qu'il faille fournir un id d'évènement. Pour accéder au héros, il faut utiliser l'id « 0 » (zéro).

Lorsqu'un argument possède une valeur par défaut, cela veut dire qu'il n'est pas obligatoirement spécifiable. Un argument par défaut est précédé d'une

étoile (*) dans le titre et il y a sa valeur indiquée, par exemple :
color(rouge vert, bleu, *alpha=0) indique que alpha n'est pas obligatoirement spécifiable et qu'il vaut par défaut (donc s'il n'est pas ajouté) 0.
Cependant, il faut toujours respecter l'ordre des arguments ce qui veut dire que si je possède une commande qui possède 5 arguments dont 3 par défauts (toujours les 3 derniers) et que je veux spécifier une valeur pour le 4ème, je devrais aussi spécifier une valeur pour le 3ème. Seuls les arguments de fin de commande peuvent utiliser une valeur par défaut.

Lorsqu'une commande n'est pas rédigée correctement (faute de frappe par exemple), l'interpréteur RPGMaker indiquera qu'il y a une erreur et suggérera la bonne orthographe.

Commandes standards

Dans cette partie nous allons expliquer les commandes dites « standard », soit celles qui sont très souvent utilisées mais qui ne rentre pas spécialement dans une catégorie.

random(nombre1, nombre2)

```
random(10, 25)  
cmd(:random, 10, 25)
```

Donne un nombre entier aléatoire compris entre nombre1 et nombre2 (compris).

Cette commande est identique a celle des évènements standards de RPG Maker. Elle est « plus pratique » dans le sens où il est possible de lui donner ce qu'on veut comme nombre en argument, ce qui veut dire qu'on peut lui passer des variables locales ou des variables globales.

Exemple simple

Voici un mini jeu où il faut découvrir un nombre secret, à chaque fois qu'un nombre est proposé par le joueur, le programme dit si ce nombre est plus petit ou plus grand que le nombre à deviner.

Dans cet exemple j'utilise une variable locale pour stocker le nombre aléatoire mais ce n'est évidemment pas obligatoire.

Voici donc un exemple de l'utilisation de la commande random.

```

@>Afficher un message: Aucun portrait, Normal, Bas
:                               : Je choisis un nombre secret compris entre 1 et 100 !
:                               : Essaie de le trouver !
@>Script: SV[1] = random(1, 100)
@>Boucle
  @>Attendre 1 frames
  @>Afficher un message: Aucun portrait, Normal, Bas
  :                               : Choisi un nombre !
  @>Entrer un nombre: [0001], 3 chiffre(s)
  @>Condition: Script: SV[1] == V[1]
    @>Sortir de la boucle
    @>
    : Fin de condition
  @>Condition: Script: SV[1] > V[1]
    @>Afficher un message: Aucun portrait, Normal, Bas
    :                               : Mon nombre est plus grand !
    @>
    : Fin de condition
  @>Condition: Script: SV[1] < V[1]
    @>Afficher un message: Aucun portrait, Normal, Bas
    :                               : Mon nombre est plus petit !
    @>
    : Fin de condition
  @>
  : Fin de boucle
@>Afficher un message: Aucun portrait, Normal, Bas
:                               : Bravo ! Tu as trouvé !
:                               : On recommence quand tu veux !

```

Je l'ai placé dans un évènement simple avec une apparence.
Je ne pense pas que ce code crée d'ambiguïté, il est très naïf et très simple.

map_id

```
map_id
```

```
cmd(:map_id)
```

Donne l'id de la map.

Cette commande n'est pas digne d'un grand intérêt, son seul intérêt réside dans le fait qu'elle puisse être assignée à une variable locale.

percent(valeur, maximum)

```
percent(7, 14)
```

```
cmd(:percent, 7, 14)
```

Convertit en % le rapport entre « valeur » et « maximum »

Dans l'exemple donné, le résultat vaudra 50 parce que 7 correspond à 50 % de 14.

Exemple simple

Nous allons améliorer notre petit jeu de la devinette. Nous allons compter le

nombre de parties gagnées, le nombre de parties perdues et donner un pourcentage de réussite du jeu. Pour ça il faudra qu'on mette une limite de nombre de choix. Je propose 10 essais avant de déclarer la partie perdue. Le code est assez simple. On ajoute une variable globale qui compte le nombre de parties, une variable locale qui compte le nombre d'essais et une variable globale qui compte le nombre de parties gagnées. Une fois qu'on sort de la boucle, on calcule le pourcentage de réussite au moyen de la commande percent.

```
@>Afficher un message: Aucun portrait, Normal, Bas
:
: Je choisis un nombre secret compris entre 1 et 100 !
:
: Essaie de le trouver !
@>Script: SV[1] = random(1, 100)
:
: V[2] += 1
:
: SV[2] = 0
@>Boucle
@>Attendre 1 frames
@>Condition: Script: SV[2] == 10
@>Afficher un message: Aucun portrait, Normal, Bas
:
: Aie, tu as perdu, le nombre était \SV[1]
@>Sortir de la boucle
@>
: Fin de condition
@>Afficher un message: Aucun portrait, Normal, Bas
:
: Choisi un nombre !
@>Entrer un nombre: [0001], 3 chiffre(s)
@>Condition: Script: SV[1] == V[1]
@>Script: V[3] += 1
@>Afficher un message: Aucun portrait, Normal, Bas
:
: Bien joué !
@>Sortir de la boucle
@>
: Fin de condition
@>Script: SV[2] += 1
@>Condition: Script: SV[1] > V[1]
@>Afficher un message: Aucun portrait, Normal, Bas
:
: Mon nombre est plus grand !
@>
: Fin de condition
@>Condition: Script: SV[1] < V[1]
@>Afficher un message: Aucun portrait, Normal, Bas
:
: Mon nombre est plus petit !
@>
: Fin de condition
@>
: Fin de boucle
@>Script: SV[3] = percent(V[3], V[2])
@>Afficher un message: Aucun portrait, Normal, Bas
:
: Tu as \SV[3]% de réussite !
@>
```

J'alterne entre des variables locales et globales pour insister sur le fait que leur utilisation est très similaire.

Comme vous pouvez le voir, il est très facile d'afficher une variable locale dans les messages.

Vous auriez pu utiliser la commande standard pour modifier les variables globales mais je trouve ça plus rapide d'utiliser un appel de script. C'est

vraiment un choix personnel.

proportion(pourcent, maximum)

```
proportion(50, 14)
```

```
cmd(:proportion, 50, 14)
```

Applique le pourcentage à un maximum (retourne ce résultat).

Dans les deux exemples, la variable 1 contiendra 7, parce que 50 % de 14 = 7.

Cette commande est très utilisée avec la commande percent, parce qu'elle permet de très rapidement faire des règles de trois.

On calcule un pourcentage qu'on applique sur un autre maximum.

color(rouge, vert, bleu, *alpha=0)

```
color(0 255,0)
```

```
cmd(:color, 0, 255, 120, 50)
```

Crée une couleur (à passer en argument par exemple) via une valeur pour le rouge (R), pour le vert (G), et pour le bleu (B), A (Alpha) est par défaut à 255 (l'opacité) donc elle n'est pas obligatoirement spécifiée.

Cette commande sert pour les cas où il faut passer une couleur en argument d'une commande. Les valeurs vont de zéro à 255. (Plus une couleur est proche de 255 au plus elle sera claire).

Cette commande n'est que très rarement utile, elle permet simplement, dans certains cas, de passer facilement une couleur en argument.

flash_square(coordonnées, r, g, b)

Fait clignoter une case de la map d'une certaine couleur si un caractère est dessus.

Cette commande fonctionne un peu particulièrement parce qu'il est possible de lui donner plein d'arguments différents. Les trois derniers arguments seront toujours des valeurs pour définir la couleur de clignotement, en revanche, les coordonnées peuvent être exprimées sous plusieurs formes. Nous allons les voir une par une.

La première méthode est celle qui va faire clignoter une case. Voici un exemple avec un clignotement rouge :

- *flash_square([2,2], 255, 0, 0)*
- *flash_square(2,2,255,0,0)*

Ces deux manières produisent le même résultat, ils flasheront la case 2,2 de la carte.

Il est aussi possible de passer deux structures énumérables (mais qui doivent avoir la même taille !) pour flasher des grandes zones :

- *flash_square([2,3,4,5],[2,3,4,5], 255, 0, 0)*
- *flash_square((0..10),(0..10), 255, 0, 0)*

Notez que ces « .. » sont réellement utilisables.

Le premier des deux arguments correspondra aux 'x' et le second aux 'y'.
Il est aussi possible de créer une ligne de case flashée en mettant un nombre pour un des deux arguments.

- `flash_square([1,2,3,4,5], 10, 255, 0, 0)`
- `flash_square(10, (11..28), 255, 0, 0)`

Voici un petit exemple d'utilisation (dont l'utilité est assez douteuse, je l'admet) , qui permet de flasher chaque case sur laquelle le héros va (un évènement en processus parallèle):

```
@>Attendre 1 frames
@>Variable: [0005] = Héros Carte X
@>Variable: [0006] = Héros Carte Y
@>Script: cmd(:flash_square, V[5], V[6], 255, 0, 0)
@>
```

Cette commande peut être utile dans la réalisation d'un Tactical RPG, par exemple.

unflash_square(coordonnées)

Arrête le clignotement d'une case.

Cette commande arrête de faire clignoter une case. Elle prend exactement le même type de coordonnées que la commande `flash_square`, il est donc aussi possible de lui passer des énumérables pour gérer des zones. Il n'existe pas de `unflash_all` mais il est possible d'utiliser `unflash` sur l'ensemble de la map.

square_passable?(x, y, *direction = 2)

```
square_passable?(10, 20)
cmd(:square_passable?, 10, 20, 6)
```

Renvoie true si les coordonnées données sont passables (via la direction passée en argument qui vaut par défaut 2 (2, 4, 6, 8 (bas, gauche, droite, haut).). Sinon, la commande renvoie false.

Commandes systèmes

Ces commandes comprennent les commandes d'information de système du jeu. Elles sont identiques à celles comprises dans les opérandes de variables RPG Maker, donc si elles existent, c'est pour pouvoir être liées à des variables locales (par exemple).

team_size

```
team_size
cmd(:team_size)
```

Donne la taille de l'équipe.

gold

```
gold  
cmd(:gold)
```

Donne la quantité d'argent possédé par l'équipe.

steps

```
steps  
cmd(:steps)
```

Donne le nombre de pas effectués par l'équipe.

play_time

```
play_time  
cmd(:play_time)
```

Donne la durée de jeu.

timer

```
timer  
cmd(:timer)
```

Donne la valeur courante du chronomètre.

save_count

```
save_count  
cmd(:save_count)
```

Donne le nombre de sauvegarde effectuées sur la partie.

battle_count

```
battle_count  
cmd(:battle_count)
```

Donne le nombre de combats effectués.

Commandes d'opérandes sur les objets

Utilitaires pour compter le nombre d'objets (d'une espèce d'objet précise, pas l'ensemble) possédé. (Identique aux commandes événementielles natives). Leur principal intérêt est donc de pouvoir être facilement liés à une variable globale. Ne pas confondre avec les objets possédés par un héros particulier. Elles sont identiques aux commandes événementielles de VX Ace (mais on peut leur passer des arguments sous forme de variables, variables locales etc.)

item_count(id_objet)

```
item_count(10)
```



```
cmd(:item_count, 10)
```

Donne le nombre d'un certain objet possédé par l'équipe.
Objet défini en fonction de son ID.

weapon_count(id_arme)

```
weapon_count(10)
```

```
cmd(:weapon_count, 10)
```

Donne le nombre d'une certaine arme possédé par l'équipe.
Arme définie en fonction de son ID.

armor_count(id_armure)

```
armor_count(10)
```

```
cmd(:armor_count, 10)
```

Donne le nombre d'une certaine armure possédé par l'équipe.
Armure définie en fonction de son ID.

item_name(id_objet)

```
item_name(10)
```

```
cmd(:item_name, 10)
```

Donne le nom d'un objet en fonction de son ID.

weapon_name(id_arme)

```
weapon_name(10)
```

```
cmd(:weapon_name, 10)
```

Donne le nom d'une arme en fonction de son ID.

armor_name(id_armure)

```
armor_name(10)
```

```
cmd(:armor_name, 10)
```

Donne le nom d'une armure en fonction de son ID.

Commandes de partie

Dans cette section nous allons voir les commandes qui font référence à la partie.

actor_level(id_actor)

```
actor_level(1)
```

```
cmd(:actor_level, 1)
```

Donne le niveau d'un membre de l'équipe en fonction de son ID.

actor_experience(id_actor)

```
actor_experience(1)
```

```
cmd(:actor_experience, 1)
```

Donne l'expérience d'un membre de l'équipe en fonction de son ID.

Il est aussi possible d'utiliser « actor_exp(id) ».

actor_hp(id_actor)

```
actor_hp(1)
```

```
cmd(:actor_hp, 1)
```

Donne le nombre de points de vie d'un membre de l'équipe en fonction de son ID.

actor_mp(id_actor)

```
actor_mp(1)
```

```
cmd(:actor_mp, 1)
```

Donne le nombre de points de magie d'un membre de l'équipe en fonction de son ID.

actor_max_hp(id_actor)

```
actor_max_hp(1)
```

```
cmd(:actor_max_hp, 1)
```

Donne le nombre de points de vie maximum (quand le membre a 100 % de vie) d'un membre de l'équipe en fonction de son ID. Il est aussi possible d'utiliser « actor_mhp(ID) ».

actor_max_mp(id_actor)

```
actor_max_mp(1)
```

```
cmd(:actor_max_mp, 1)
```

Donne le nombre de points de magie maximum (quand le membre a 100 % de magie) d'un membre de l'équipe en fonction de son ID. Il est aussi possible d'utiliser « actor_mmp(ID) ».

actor_attack(id_actor)

```
actor_attack(1)
```

```
cmd(:actor_attack, 1)
```

Donne le nombre de points d'attaque d'un membre de l'équipe en fonction de son ID, il est aussi possible d'utiliser « actor_atk(id) ».

actor_defense(id_actor)

```
actor_defense(1)
```

```
cmd(:actor_defense, 1)
```

Donne le nombre de points de défense d'un membre en fonction de son ID, il est aussi possible d'utiliser « actor_def(id) ».

actor_magic(id_actor)

```
actor_magic(1)
```

```
cmd(:actor_magic, 1)
```

Donne le nombre de points d'attaque magique d'un membre en fonction de son ID. Il est aussi possible d'utiliser :

- actor_magi(id)
- actor_mag(id)
- actor_mat(id)

actor_magic_defense(id_actor)

```
actor_magic_defense(1)
```

```
cmd(:actor_magic_defense, 1)
```

Donne le nombre de points de magie défensive d'un membre en fonction de son ID, il est aussi possible d'utiliser « actor_mdf(id) ».

actor_agility(id_actor)

```
actor_agility(1)
```

```
cmd(:actor_agility, 1)
```

Donne le nombre de points d'agilité d'un membre en fonction de son ID. Il est aussi possible d'utiliser « actor_agi(id) ».

actor_luck(id_actor)

```
actor_luck(1)
```

```
cmd(:actor_luck, 1)
```

Donne le nombre de points de chance d'un membre en fonction de son ID. Il est aussi possible d'utiliser « actor_luk(id) ».

actor_name(id_actor)

```
actor_name(1)
```

```
cmd(:actor_name, 1)
```

Renvoie le nom d'un membre en fonction de son ID.

set_actor_name(id_actor, nouveau_nom)

```
set_actor_name(1, "Grim")
```

```
cmd(:actor_name, 1, "Grim")
```

Change le nom du membre (référéncé par son ID) par le nom passé en

argument.(ici, l'ancien nom est remplacé par Grim)

Commandes de lecture de données des monstres

Contrairement aux commandes event-making traditionnelles, ici, il est possible d'accéder aux données des monstres (définies dans la base de données).

read_monster_data(id_monstre, donnee_voulue)

```
read_monster_data(1,:hp)
cmd(:read_monster_data, 1,:mp)
```

Permet d'accéder aux informations statiques (contenue dans la base de données) du monstre.

La liste des données récupérables est :

- :hp
- :mp
- :attack
- :defense
- :magic_attack
- :magic_defense
- :agility
- :luck

Sinon, il est possible d'accéder à chacune de ces données via leurs commandes respectives qui sont décrites ci-dessous.

monster_hp(id_monstre)

```
monster_hp(1)
cmd(:monster_hp, 1)
```

Donne le nombre de points de vie d'un monstre en fonction de son ID.

monster_mp(id_monstre)

```
monster_mp(1)
cmd(:monster_mp, 1)
```

Donne le nombre de points de magie d'un monstre en fonction de son ID.

monster_attack(id_monstre)

```
monster_attack(1)
cmd(:monster_attack, 1)
```

Donne le nombre de points d'attaque d'un monstre en fonction de son ID. Il est aussi possible d'utiliser « actor_atk(id) ».

monster_defense(id_monstre)

```
monster_defense(1)
```

```
cmd(:monster_defense, 1)
```

Donne le nombre de points de défense d'un monstre en fonction de son ID. Il est aussi possible d'utiliser « actor_def(id) ».

monster_magic_attack(id_monstre)

```
monster_magic_attack(1)
```

```
cmd(:monster_magic_attack, 1)
```

Donne le nombre de points d'attaque magique d'un monstre en fonction de son ID. Il est aussi possible d'utiliser « actor_mat(id) ».

monster_magic_defense(id_monstre)

```
monster_magic_defense(1)
```

```
cmd(:monster_magic_defense, 1)
```

Donne le nombre de points de défense magique d'un monstre en fonction de son ID. Il est aussi possible d'utiliser « actor_mdf(id) ».

monster_agility(id_monstre)

```
monster_agility(1)
```

```
cmd(:monster_agility, 1)
```

Donne le nombre de points d'agilité d'un monstre en fonction de son ID. Il est aussi possible d'utiliser « actor_agi(id) ».

monster_luck(id_monstre)

```
monster_luck(1)
```

```
cmd(:monster_luck, 1)
```

Donne le nombre de points de chance d'un monstre en fonction de son ID. Il est aussi possible d'utiliser « actor_luk(id) ».

monster_name(id_monstre)

```
monster_name(1)
```

```
cmd(:monster_name, 1)
```

Donne le nom d'un monstre en fonction de son ID.

troop_size(id_groupe_monstre)

```
troop_size(1)
```

```
cmd(:troop_size, 1)
```

Donne la taille d'un groupe de monstres (soit le nombre de monstres présents dans le groupe), tels que définis dans la base de données.

troop_member_id(id_groupe_monstre, position)

```
troop_member_id(1, 2)
```

```
cmd(:troop_member, 1, 2)
```

Donne l'ID du monstre dans un groupe référencé par son ID en fonction de sa position.

Par exemple, dans cet exemple :



Les positions 1 et 2 retourneront l'ID du slime et la position 3 retournera l'identifiant de l'araignée.

troop_member_x(id_groupe_monstre, position)

```
troop_member_x(1, 2)
```

```
cmd(:troop_member_x, 1, 2)
```

Retourne la position (en X) du monstre d'un groupe référencé par son ID en fonction de sa position, tel que défini dans la base de données.

troop_member_y(id_groupe_monstre, position)

```
troop_member_y(1, 2)
```

```
cmd(:troop_member_y, 1, 2)
```

Retourne la position (en Y) du monstre d'un groupe référencé par son ID en fonction de sa position, tel que défini dans la base de données.

Ces deux commandes (x et y) permettent de positionner des monstres, tels que défini dans la base de données.

Commandes de manipulation de la souris

Manipuler la souris est une chose impossible à réaliser en Event Making traditionnel, l'Event Extender offre une interface complète de manipulation de souris.

mouse_x

```
mouse_x
```

```
cmd(:mouse_x)
```

Donne la position X de la souris sur l'écran.

mouse_y

```
mouse_y
```

```
cmd(:mouse_y)
```

Donne la position Y de la souris sur l'écran.

Exemple simple

Avec ce couple de commandes, il est très simple de créer rapidement un système de curseur :

```
@>Afficher une image: n°1, '001-Weapon01', origine: haut-gauche (variable [0001][0002]),
@>Boucle
  @>Attendre 1 frames
  @>Script: V[1], V[2] = mouse_x, mouse_y
  @>Déplacer une image: n°1, origine: haut-gauche (Variable [0001][0002]), zoom: 100%
  @>
: Fin de boucle
@>
```

Il suffit de placer l'image en fonction de variables, qui elles-mêmes prennent le X et le Y de la souris.

Cet exemple est un événement en processus parallèle.

mouse_x_square

```
mouse_x_square
cmd(:mouse_x_square)
```

Donne la coordonnée X de la case sur laquelle se trouve la souris.

mouse_y_square

```
mouse_y_square
cmd(:mouse_y_square)
```

Donne la coordonnée Y de la case sur laquelle se trouve la souris.

Exemple simple

Ce couple de commandes permet de détecter la case (et non le pixel) sur laquelle se trouve la souris. Il peut être très utile pour faire un système de sélection de personnage.

Notre exemple utilise la commande `flash_square` pour faire clignoter en vert la case sur laquelle la souris est passée :

```
@>Attendre 1 frames
@>Script: flash_square(mouse_x_square, mouse_y_square, 0, 255, 0)
@>
```

J'ai directement placé les commandes dans les arguments de `flash_square`. Cet exemple est aussi un événement en processus parallèle.

Liste des boutons cliquables

Nous allons maintenant nous intéresser aux détections de clics de la souris. Pour ça, nous avons trois boutons définis :

- :mouse_left
- :mouse_right
- :mouse_center

Il s'agit des noms que l'on peut passer en argument dans les cas où un bouton est demandé.

mouse_click?(bouton)

```
mouse_click?(:mouse_left)
cmd(:mouse_click?, :mouse_left)
```

Renvoie true si le bouton passé en argument est enfoncé, false sinon. Tant que la touche passée en argument est enfoncée, la commande renverra true.

Exemple simple

Voici un petit événement en processus parallèle qui fera sauter le héros chaque fois que le bouton central de la souris sera enfoncé :

```
@>Boucle
  @>Attendre 1 frames
  @>Condition: Script: mouse_click?(:mouse_center)
    @>Déplacer un évènement: Héros (attendre la fin)
      :                               : $>Sauter: +0,+0
    @>
      : Fin de condition
  @>
  : Fin de boucle
```

En général, lorsque je dois détecter des choses (comme des pressions de touches), j'utilise, dans mon événement, une boucle (avec un Attendre 1 frames pour éviter le lag) car il me semble que la détection est plus précise. Mais je vous invite à faire vos propres expériences.

Nous allons voir qu'il existe d'autres solutions de détection de touches.

mouse_trigger?(bouton)

```
mouse_trigger?(:mouse_left)
cmd(:mouse_trigger?, :mouse_left)
```

Renvoie true si le bouton passé en argument est cliqué, false sinon. Contrairement à mouse_click?, la détection n'est pas continue mais « directe », elle fonctionne comme la fonction « Input.trigger? » de RPG Maker. Si on maintient le clic enfoncé pendant 15 secondes, et qu'on teste la valeur de mouse_trigger? Elle ne vaudra true qu'une seule fois.

mouse_repeat?(bouton)

```
mouse_repeat?(:mouse_left)
cmd(:mouse_repeat?, :mouse_left)
```

Renvoie true si le bouton passé en argument est appuyé successivement. Son comportement est identique à Input.repeat?

mouse_release?(bouton)

```
mouse_release?(:mouse_left)
cmd(:mouse_release?, :mouse_left)
```

Renvoie true si le bouton passé en argument vient d'être relâché, false sinon.

Note sur les type de détections

Qu'il s'agisse de la souris ou quand nous le verrons, du clavier, je vous invite à essayer les différentes commandes de détections pour en saisir la nuance. Personnellement, je me sers de trigger quand je dois détecter un clique, de click quand je dois détecter une pression continue et rarement des autres.

point_in_mouse_rect?(x, y)

```
point_in_mouse_rect?(10, 20)
cmd(:point_in_mouse_rect?, 200, 150)
```

Renvoie true si le point (référéncé par x et y) est compris dans le dernier rectangle de sélection effectué par la souris.

mouse_hover_event?(id_event)

```
mouse_hover_event?(10)
cmd(:mouse_hover_event?, 10)
```

Renvoie true si la souris survole un événement référencé par son ID. Rappelons que le héros est référencé par l'id 0 (zéro).

Exemple simple

Voici un petit exemple à mettre dans un événement en processus parallèle si l'on veut qu'au survol de ce dernier, il se mette à sauter.

```
@>Condition: Script: mouse_hover_event?(@event_id)
  @>Déplacer un événement: Cet événement (attendre la fin)
  :                               : $>Sauter: +0,+0
  @>
  : Fin de condition
  @>
```

Dans cet exemple on peut voir l'utilisation de « @event_id », qui permet de récupérer l'id de l'évènement courant. Soit l'id de l'évènement dans lequel on se trouve.

mouse_clicked_event?(id_event, bouton)

```
mouse_clicked_event?(10, :mouse_right)
```

```
cmd(:mouse_clicked_event?, 10, :mouse_right)
```

Renvoie true si la souris clique un événement référencé par son ID, false sinon. Rappelons que le héros est référencé par l'id 0 (zéro). La détection est effectuée si la case où se trouve l'événement est cliquée.

mouse_hover_player?

```
mouse_hover_player?
```

```
cmd(:mouse_hover_player?)
```

Renvoie true si la souris survole le héros.

mouse_clicked_player?(bouton)

```
mouse_clicked_player?(:mouse_center)
```

```
cmd(:mouse_clicked_player?, :mouse_center)
```

Renvoie true si la souris survole le héros.

show_cursor_system(value)

```
show_cursor_system(true)
```

```
cmd(:show_cursor_system, false)
```

Si l'argument value vaut true, le curseur de Windows sera affiché, si il vaut false, il ne sera pas affiché. (Uniquement pour le mode de jeu fenêtré, en plein écran le curseur de Windows n'est jamais affiché).

Commandes de manipulation du clavier

Le nombre de touches manipulables dans RPG Maker est limité, avec l'Event Extender il est possible d'en manipuler quelques unes en plus. Voici la liste des touches que l'on peut utiliser (soit à passer en argument quand une touche est demandée en argument).

Liste des touches du clavier prises en charge par l'Event Extender

- :tab
- :backspace
- :clear
- :enter
- :shift
- :ctrl
- :alt
- :pause
- :caps_lock

- :esc
- :space
- :page_up
- :page_down
- :end
- :home
- :left
- :up
- :right
- :down
- :select
- :print
- :execute
- :help
- :zero
- :one
- :two
- :three
- :four
- :five
- :six
- :seven• :eight
- :nine
- :minus
- :a
- :b
- :c
- :d
- :e
- :f
- :g
- :h
- :i
- :j

- :k
- :l
- :m
- :n
- :o
- :p
- :q
- :r
- :s
- :t
- :u
- :v
- :w
- :x
- :y
- :z
- :lwindow
- :rwindow
- :apps
- :num_zero
- :num_one
- :num_two
- :num_three
- :num_four
- :num_five
- :num_six
- :num_seven• :num_eight
- :num_nine
- :multiply
- :add
- :subtract
- :decimal
- :divide
- :f1

- :f2
- :f3
- :f4
- :f5
- :f6
- :f7
- :f8
- :f9
- :f10
- :f11
- :f12
- :num_lock
- :scroll_lock
- :lshift
- :rshift
- :lcontrol
- :rcontrol
- :lmenu
- :rmenu
- :circumflex
- :dollar
- :close_parenthesis
- :u_grav
- :square
- :less_than
- :colon
- :semicolon
- :equal
- :comma

key_press?(bouton)

```
key_press?(:space)
```

```
cmd(:key_press?, :enter)
```

Renvoie true si la touche passée en argument est enfoncée, false sinon.

key_trigger?(bouton)

```
key_trigger?(:space)  
cmd(:key_trigger?, :enter)
```

Renvoie true si la touche passée en argument est appuyée, false sinon.
Contrairement à key_press?, la détection n'est pas continue mais « directe », elle fonctionne comme la fonction « Input.trigger? » de RPG Maker. (mais avec toutes les touches énoncées plus haut).

Nuances entre Trigger? Et press?

Press : Détermine si le bouton correspondant au symbole sym est en train d'être appuyé ou non.

Renvoie TRUE si le bouton est appuyé, renvoie FALSE sinon.

Trigger : Détermine si le bouton correspondant au symbole sym est train d'être appuyé une nouvelle fois.

Un bouton est "Appuyé une nouvelle fois" si du temps est passé entre l'état "non appuyé" et l'état "appuyé" du bouton.

Renvoie TRUE si le bouton est en train d'être appuyé, renvoie FALSE sinon.

key_repeat?(bouton)

```
key_repeat?(:space)  
cmd(:key_repeat?, :enter)
```

Renvoie true si le bouton passé en argument est appuyé successivement. Son comportement est identique à Input.repeat? De RPG Maker. (mais avec toutes les touches énoncées plus haut).

key_release?(bouton)

```
key_release?(:space)  
cmd(:key_release?, :enter)
```

Renvoie true si le bouton passé en argument était appuyé et vient d'être relâché (false sinon).

caps_lock?

```
caps_lock?  
cmd(:caps_lock?)
```

Renvoie true si le clavier est verrouillé en majuscule (CAPS_LOCK enfoncée), false sinon.

num_lock?

```
num_lock?  
cmd(:num_lock?)
```

Renvoie true si le pavé numérique est activé, false sinon.

scroll_lock?

```
scroll_lock?  
cmd(:scroll_lock?)
```

Renvoie true si la touche SCROLL_LOCK est verrouillée, false sinon.

maj?

```
maj?  
cmd(:maj?)
```

Renvoie true si le clavier est en mode majuscule (donc si la touche maj est enfoncée, ou caps_lock etc.)

alt_gr?

```
alt_gr?  
cmd(:alt_gr?)
```

Renvoie true si la combinaison Alt+gr (ou ctrl + alt) est enfoncée.

key_number

```
key_number  
cmd(:key_number)
```

Renvoie le chiffre enfoncé actuellement pressé (via les touches en haut du clavier ou du pavé numérique). Si aucune touche n'est enfoncée, la commande retournera la valeur « nil » (qui correspond à « rien » en Ruby).

key_char

```
key_char  
cmd(:key_char)
```

Renvoie le caractère correspondant à celui pressé au clavier (chiffres compris), il gère « a priori » tous les caractères (y compris ñ etc.) Si aucun caractère n'est pressé, la commande renvoie une chaîne de caractères vide (soit "").

Exemple simple

Voici un rapide exemple qui affiche dans un message le caractère correspondant à celui pressé sur le clavier.

```
@>Boucle  
  @>Attendre 1 frames  
  @>Variable: [0001] = key_char  
  @>Condition: Script: V[1] != ""  
    @>Afficher un message: Aucun portrait, Normal, Bas  
    :                      : Caractère pressé : \V[1]  
    @>  
  : Fin de condition  
  @>  
: Fin de boucle  
@>
```

Si le caractère « ^ » est pressé avant un « a », le caractère généré sera « â », mais si, par exemple, un « p » est pressé ensuite, le caractère généré sera « ^p ».

Commandes de manipulation des évènements

L'objectif de cette collection de commandes est d'offrir plus d'informations et d'outils sur la manipulation des évènements.

Il a pour objectif de rendre « simples », certaines opérations longues à réaliser en Event Making classique (comme par exemple la distance entre deux évènements, vérifier si deux évènements sont en collision, vérifier si un évènement est visible à l'écran, déplacer des évènements etc.).

event_x(id)

```
event_x(10)  
cmd(:event_x, 10)
```

Donne la position x (en cases) d'un évènement en fonction de son ID. Pour rappel, si l'ID zéro est fourni, la commande s'appliquera au héros. (Et ceci est pareil pour toutes les commandes qui suivent).

event_y(id)

```
event_y(10)  
cmd(:event_y, 10)
```

Donne la position y (en cases) d'un évènement en fonction de son ID.

event_screen_x(id)

```
event_screen_x(10)  
cmd(:event_screen_x, 10)
```

Donne la position x (en pixels par rapport à l'écran) d'un évènement en fonction de son ID.

event_screen_y(id)

```
event_screen_y(10)  
cmd(:event_screen_y, 10)
```

Donne la position y (en pixels par rapport à l'écran) d'un évènement en fonction de son ID.

event_pixel_x(id)

```
event_pixel_x(10)
```

```
cmd(:event_pixel_x, 10)
```

Donne la position y (en pixels par rapport à l'origine de la carte) d'un évènement en fonction de son ID.

event_pixel_y(id)

```
event_pixel_y(10)
```

```
cmd(:event_pixel_y, 10)
```

Donne la position y (en pixels par rapport à l'origine de la carte) d'un évènement en fonction de son ID.

event_direction(id)

```
event_direction(10)
```

```
cmd(:event_direction, 10)
```

Donne la direction d'un évènement en fonction de son ID.

Pour rappel, les valeurs retournées sont 2, 4, 6, 8 (bas, gauche, droite, haut).

player_x

```
player_x
```

```
cmd(:player_x)
```

Donne la position x (en cases) du joueur. (identique à event_x(0), Cependant cette version est plus compressée:))

player_y

```
player_y
```

```
cmd(:player_y)
```

Donne la position y (en cases) du joueur. (identique à event_y(0)).

player_screen_x

```
player_screen_x
```

```
cmd(:player_screen_x)
```

Donne la position x (en pixels par rapport à l'écran) du joueur. (identique à event_screen_x(0))

player_screen_y

```
player_screen_y
```

```
cmd(:player_screen_y)
```

Donne la position y (en pixels par rapport à l'écran) du joueur. (identique à event_screen_y(0))

player_direction

```
player_direction
```

```
cmd(:player_direction)
```

Donne la direction du joueur. (identique à event_direction(0))

distance_between(type, event1, event2)

```
distance_between(:pixel, 2, 3)
```

```
cmd(:distance_between,:square, 2, 3)
```

Donne la distance (à vol d'oiseau) entre deux évènements (pour rappel, l'event 0 correspond au joueur). Le type peut prendre deux valeurs :

1. :pixel
2. :square

:pixel donnera le nombre de pixel entre les deux évènements et :square le nombre de cases. Le résultat sera toujours envoyé sous la forme d'un entier.

look_at(event1, event2, scope, *metric=:square)

```
look_at(1, 2, 40,:pixel)
```

```
cmd(:look_at, 2, 3, 2)
```

Renvoie true si l'event 1 regarde l'event 2 (donc regarde dans la direction de l'event 2 et que l'event 2 est situé à une moins grande distance que celle définie dans scope, en changeant la valeur de metric, il est possible de spécifier la portée en pixel ou en case.

squares_between(event1, event2)

```
squares_between(1, 2)
```

```
cmd(:squares_between, 1, 2)
```

Donne la distance en cases entre deux évènements, est identique à distance_between(:square, event1, event2).

pixels_between(event1, event2)

```
pixels_between(1, 2)
```

```
cmd(:pixels_between, 1, 2)
```

Donne la distance en pixels entre deux évènements, est identique à distance_between(:pixel, event1, event2).

Exemple simple

Nous allons voir ici comment comportementaliser rapidement un évènement pour le faire suivre le héros si celui-ci s'approche de moins de 3 cases. Si le héros prend de la distance, soit plus de 3 cases, notre évènement arrête de suivre le héros :

```

@>Attendre 1 frames
@>Condition: Script: squares_between(0,1) <= 3
    @>Déplacer un évènement: Cet évènement (attendre la fin)
    :                               : $>Un pas vers le héros
    @>
    : Fin de condition
@>

```

Dans un évènement en processus parallèle:)

move_to(event_id, x, y, *wait=false)

```

move_to(0, 10, 10)
cmd(:move_to, 0, 10, 10, true)

```

Cette commande déplace un évènement référencé par son ID jusqu'aux coordonnées passées en argument (x et y). Le dernier argument est facultatif, il permet de dire si l'évènement **doit** avoir fini son trajet avant de pouvoir effectuer une autre action. Par défaut, le « wait » est désactivé.

Mise en garde

Cette commande utilise un algorithme relativement gourmand, il ne faut donc pas en abuser (par exemple, recherche de chemin dans un labyrinthe sur une carte de 500 x 500).

Exemple simple

Voici un tout petit point&click minimaliste (car nous ne nous soucions pas de l'interaction avec les autres évènements) pour déplacer le joueur vers les emplacements cliqués par la souris (Utilisation des commandes vues dans le module Souris!)

```

@>Boucle
    @>Attendre 1 frames
    @>Condition: Script: mouse_trigger?(:mouse_left)
        @>Script: move_to(0, mouse_x_square, mouse_y_square)
        @>
        : Fin de condition
    @>
    : Fin de boucle
@>

```

Le tout dans un évènement parallèle. Il serait aussi possible d'imaginer une toute petite variante où la case serait flashée avant (pour un peu plus de visibilité:)).

L'idée générale de l'évènement est de flasher la case sur laquelle la souris se trouve et de la déflasher si la case où se trouvait la souris a changé.

Nous allons aussi utiliser la commande « square_passable? » pour que la case soit flashée en rouge si la case est non passable et en vert si la case est

passable !

Donc je joue avec des variables locales. Je ne suis pas sûr que ce code soit le plus « optimisé » possible mais il a, je trouve, le mérite d'être clair et facile à comprendre :

```
@>Boucle
  @>Attendre 1 frames
  @>Condition: Script: SV[1] != mouse_x_square or SV[2] != mouse_y_square
    @>Script: unflash_square(SV[1], SV[2])
    @>Script: SV[1], SV[2] = mouse_x_square, mouse_y_square
    @>Condition: Script: square_passable?(SV[1], SV[2])
      @>Script: flash_square(SV[1], SV[2], 0, 255, 0)
      @>
    : Sinon
      @>Script: flash_square(SV[1], SV[2], 255, 0, 0)
      @>
    : Fin de condition
  @>
  : Fin de condition
  @>Script: SV[1], SV[2] = mouse_x_square, mouse_y_square
  @>Condition: Script: mouse_trigger?(:mouse_left)
    @>Script: move_to(0, SV[1], SV[2])
    @>
  : Fin de condition
  @>
  : Fin de boucle
```

Une fois de plus dans un évènement en processus parallèle.

jump_to(event_id, x, y, *wait=true)

```
jump_to(1, 15, 15)
cmd(:jump_to, 1, 15, 15, false)
```

Fait sauter l'évènement référencé par son ID vers la case référencées par x et y, par défaut, l'argument d'attente est activé pour qu'un évènement ne puisse débuter un nouveau saut sans avoir terminer son saut en cours.

Exemple simple

Il s'agit du même que l'exemple précédent (avec les cases flashées) sauf que cette fois, le héros sautera pour se rendre au lieu défini par le clic :

```

@>Boucle
  @>Attendre 1 frames
  @>Condition: Script: SV[1] != mouse_x_square or SV[2] != mouse_y_square
    @>Script: unflash_square(SV[1], SV[2])
    @>Script: SV[1], SV[2] = mouse_x_square, mouse_y_square
    @>Condition: Script: square_passable?(SV[1], SV[2])
      @>Script: flash_square(SV[1], SV[2], 0, 255, 0)
      @>
      : Sinon
      @>Script: flash_square(SV[1], SV[2], 255, 0, 0)
      @>
      : Fin de condition
    @>
    : Fin de condition
    @>Script: SV[1], SV[2] = mouse_x_square, mouse_y_square
    @>Condition: Script: mouse_trigger?(:mouse_left)
      @>Script: jump_to(0, SV[1], SV[2])
      @>
      : Fin de condition
    @>
    : Fin de boucle
  @>

```

buzz(id1, id2, id3, etc...)

```
buzz(1, 2, 3, 6, 10)
```

```
cmd(:buzz, 18, 0, 17)
```

Cette commande peut prendre un nombre variable d'ID. Elle fait tressaillir (comme dans Golden Sun) une quantité variable d'évènements.

Cette commande a été écrite par Fabien pour La Factory (et réadapté par XHTMLBoy pour FUNKYWORK&BilouCorp)

collide?(event_id_1, event_id_2)

```
collide?(0, 10)
```

```
cmd(:collide?, 2, 3)
```

Vérifie si l'événement référencé par l'id1 rentre en contact avec l'événement référencé par l'id 2. (renvoi true si oui, false si non).

Exemple simple

Voici un petit exemple amusant qui utilise buzz et collide?

```

@>Attendre 1 frames
@>Condition: Script: collide?(0, 1)
  @>Déplacer un évènement: Héros (attendre la fin)
  : $>Un pas en arrière
  @>Script: buzz(1)
  @>
  : Fin de condition
@>

```

L'évènement 1 a une apparence et chaque fois que le héros lui « rentrera dedans », le héros reculera d'un pas et l'évènement tressaillira. Cette commande peut rajouter, selon moi un plus dans la mise en scène.

event_in_screen?(event_id)

```
event_in_screen?(1)
cmd(:event_in_screen?, 1)
```

Vérifie si l'évènement référencé par l'id1 est bien visible à l'écran. (renvoi true si oui, false sinon) donc pas en dehors de ce que l'écran montre de la carte (et non s'il est caché derrière un obstacle).

player_in_screen?

```
player_in_screen?
cmd(:player_in_screen?)
```

Vérifie si le héros est bien visible à l'écran, par exemple si le défilement de la carte a fait en sorte que le héros ne soit plus visible à l'écran. (renvoi true si oui, false sinon).

Équivaut à `event_in_screen?(0)`.

Commandes de manipulation des images

Ce module donnera souvent l'apparence de faire doublon avec les commandes événementielles, mais son gros atout est que comme il s'appelle au moyen d'appel de script, il est possible de placer des commandes ou des variables (globales comme locales) partout, à chaque argument.

Il devient possible et facile de manipuler chaque image avec précision.

picture_show(id_picture, nom, *x=0, *y=0, *origin=0, *zoom_x=100, *zoom_y=100, *opacity=255 *blend_mode=0)

```
picture_show(1, "Curseur", 1, 2, [14, 17])
cmd(:picture_show, 1 "Curseur")
```

Affiche une image.

L'opacité doit être comprise entre 0 et 255 (0 = totalement transparent).

Le mode de fusion (`blend_mode`) correspond à 0 = mode normal, 1 = mode ajouter, 2 = mode soustraire.

L'origine correspond à, si elle vaut 0, l'image est positionnée avec comme point d'origine le coin haut gauche. 1 avec le centre comme point d'origine mais il est aussi possible de passer une liste [x,y] et là il est possible de préciser au pixel près les coordonnées d'origine. (Voir exemple).

Le nom correspond au nom dans le dossier Pictures. Il n'est pas nécessaire de mettre l'extension.

En effet, si on définit un nom simple, l'image sera prise dans le dossier Pictures, mais il est possible d'aller chercher les images dans d'autres dossiers.

- Battlers
- Battlebacks1

- Battlebacks2
- Parallaxes
- Titles1
- Titles2

Il est donc possible de faire :

`picture_show(1, "Parallaxes/mon_parallaxe1")` ou `picture_show(1, "Battlers/soldat", 10, 20 etc.)`

Une fois qu'une image est affichée avec cette commande, il est possible de la manipuler avec les commandes événementielles fournies par défaut dans le logiciel. Si à la place d'un nom vous utilisez le symbole `:screenshot`, votre image sera une capture d'écran du jeu (`picture_show(1,:screenshot)` par exemple).

Le nombre d'arguments de cette commande est assez conséquent, nous verrons que les commandes suivantes permettent de modifier ces arguments. Je vous conseille donc de n'utiliser que les arguments obligatoires (`id`, `nom`) et d'utiliser, ensuite les commandes suivantes pour modifier chaque propriété une par une.

picture_erase(id_picture)

```
picture_erase(1)
cmd(:picture_erase, 1)
```

Supprime l'image référencée par son ID.

picture_origin(id_picture origine)

```
picture_origin(1, 0)
cmd(:picture_origin, 1, [10, 10])
```

Change l'origine d'une image en fonction de son ID.

Il est possible de soit lui attribuer 0 et l'image aura pour origine son coin haut-gauche, soit lui attribuer 1 et l'image aura pour origine son centre, soit `[x, y]` où `x` et `y` correspondent aux pixels pour orienter l'image au pixel près.

La modification de l'origine alterera la manière dont tournera l'image (via les commandes de rotation) et sa position car c'est l'origine qui définit le point d'accroche d'une image.

picture_x(id_picture, x)

```
picture_x(1, 18)
cmd(:picture_x, 1, 200)
```

Change la position `x` de l'image référencée par son ID. Si aucun `x` n'est fourni pour (`picture_x(ID)`) alors la commande retournera la position `x` actuelle de l'image.

picture_y(id_picture, y)

```
picture_y(1, 18)
cmd(:picture_y, 1, 200)
```

Change la position y de l'image référencée par son ID. Si aucun y n'est fourni pour (picture_y(ID)) alors la commande retournera la position y actuelle de l'image.

Exemple simple

Nous allons refaire notre curseur mais avec les commandes d'affichage d'image de l'Event Extender.

L'objectif est donc d'utiliser la commande picture_show pour afficher notre image de curseur et d'utiliser les commandes picture_x et picture_y pour déplacer à chaque frame notre image aux coordonnées de la souris.

Le tout dans un évènement en processus parallèle.

```
@>Script: picture_show(1, "001-Weapon01")
@>Boucle
  @>Attendre 1 frames
  @>Script: picture_x(1, mouse_x)
  :       : picture_y(1, mouse_y)
  @>
  : Fin de boucle
@>
```

picture_position(id_picture, x, y)

```
picture_position(1, 18, 150)
```

```
cmd(:picture_position, 1, 200, 35)
```

Change la position de l'image référencée par son ID. Elle revient à appeler picture_x et picture_y mais c'est pour compresser le tout en une ligne.

Exemple simple

Reprenons notre exemple précédent en utilisant la commande picture_position.

```
@>Script: picture_show(1, "001-Weapon01")
@>Boucle
  @>Attendre 1 frames
  @>Script: picture_position(1, mouse_x, mouse_y)
  @>
  : Fin de boucle
@>
```

Ce n'est pas vraiment différent mais certains préfèrent utiliser cette forme, qui économise tout de même une ligne (wow:D)

*picture_move(id_picture, x, y, zoom_x, zoom_y, duration, *opacity=-1, *blend_type=-1, *origin=-1)*

```
picture_move(1, 10, 20, 50, 50, 120)
```

```
cmd(:picture_move, 1, 200, 35, 100, 100, 60)
```


Cette commande permet de transformer une image pendant une durée donnée (en frames). Les arguments opacité, mode de fusion et origine sont facultatifs (si aucun argument n'est donné ils resteront tels qu'ils étaient, donc s'ils ont la valeur -1 ils conservent leur valeur actuelle).

Le gros atout de cette commande est qu'elle peut prendre des variables locales ou des commandes dans chaque arguments.

picture_wave(id_picture, amplitude, speed)

```
picture_wave(1, 10, 100)
cmd(:picture_wave, 1, 20, 350)
```

Permet de faire onduler une image (référéncée par son ID) en fonction d'une amplitude et d'une vitesse (comme sur Rpg Maker 2003). Pour arrêter l'ondulation, il suffit d'appeler la commande sur l'image avec une amplitude et une vitesse de 0.

picture_opacity(id_picture, valeur)

```
picture_opacity(1, 220)
cmd(:picture_opacity, 1, 200)
```

Change l'opacité (entre 0 et 255) d'une image référencée par son ID.

picture_flip(id_picture)

```
picture_flip(1)
cmd(:picture_flip, 1)
```

Applique un effet miroir à une image, avec une axe vertical. (Dans le cas d'un axe horizontal, il suffirait de faire une rotation de 180° et appliquer l'effet picture_flip;))

picture_angle(id_picture, angle)

```
picture_angle(1, 180)
cmd(:picture_angle, 1, 90)
```

Permet de tourner une image en fonction d'un angle précis, à partir du point d'origine cette commande n'est pas a confondre avec la suivante, qui fait tourner l'image en continu, picture_angle se contente de modifier l'angle d'inclinaison de l'image.

picture_rotate(id_picture, speed)

```
picture_rotate(1, 6)
cmd(:picture_rotate, 1, -6)
```

Identique à la commande événementielle, permet de faire tourner une image à une certaine vitesse. Pour la faire tourner dans l'autre sens, il suffit de mettre une vitesse négative.

Différence entre rotate et angle

Rotate : fait tourner l'image en continu.

Angle : se contente de modifier l'angle d'inclinaison de l'image.

picture_zoom_x(id_picture, zoom_x)

```
picture_zoom_x(1, 150)
```

```
cmd(:picture_zoom_x, 1, 150)
```

Modifie le Zoom en largeur en pourcentage. Si on ne donne pas l'argument de zoom, la commande renverra la valeur de zoom_x de l'image.

Par exemple si j'affiche l'image 1 avec pour zoom en largeur 120 %, l'appel de la commande picture_zoom_x(1) (donc sans zoom) renverra 120.

picture_zoom_y(id_picture, zoom_y)

```
picture_zoom_y(1, 150)
```

```
cmd(:picture_zoom_y, 1, 150)
```

Modifie le Zoom en hauteur en pourcentage. Si on ne donne pas l'argument de zoom, la commande renverra la valeur de zoom_y de l'image

picture_zoom(id_picture, zoom_x, *zoom_y=zoom_x)

```
picture_zoom(1, 150)
```

```
cmd(:picture_zoom, 1, 150, 200)
```

Zoom en longueur et en largeur en pourcentage. Si le troisième argument n'est pas donné, il sera égal à zoom_x. Donc picture_zoom(1, 150, 150) est identique à picture_zoom(1, 150).

picture_tone(id_picture, r, g, b, *gray=0)

```
picture_tone(1, 255, 255, 12, 10)
```

```
cmd(:picture_tone, 1, 255, 123, 299)
```

Modifie la teinte d'une image en fonction de son id, via sa valeur de rouge, de vert et de bleu.

Le gris (pour dé-saturer la teinte) est par défaut à zéro.

picture_blend(id_picture, mode)

```
picture_blend(1, 1)
```

```
cmd(:picture_blend, 1, 2)
```

Modifie le mode de fusion de l'image. 0 = normal, 1 = ajouter, 2 = soustraire. Si un autre nombre est donné, la commande utilisera le mode de fusion normal.

picture_pin(id_picture)

```
picture_pin(1)
```

```
cmd(:picture_pin, 1)
```

Active le défilement avec la carte.

Si la commande est appelée sur une image, cette image sera fixée sur sa position et défilera en même temps que la carte.

Exemple simple

Petit exemple d'une lune qui se reflète dans l'eau. (j'ai utilisé un évènement en processus parallèle qui s'efface une fois qu'il a fait ce qu'il devait faire). J'affiche l'image de la lune où il faut, je la fixe avec la commande et je la fait légèrement onduler.



Le rendu n'est pas exceptionnel mais c'est parce que je n'ai pas pris une très belle lune (ou du moins, une lune qui s'adapte aux ressources).

Ceci dit, il est possible d'ajouter de très jolis détails, et, par exemple, de gérer ses ombres via des images ce qui peut être pratique pour ceux qui ont la phobies des ombres auto-générée de RPG Maker VX !

picture_detach(id_picture)

```
picture_detach(1)
```

```
cmd(:picture_detach, 1)
```

Désactive le défilement pour une image qui a été « attachée » au préalable via la commande picture_pin.

picture_shake(id_picture, power, speed, duration)

```
picture_shake(1, 10, 20, 60)
```

```
cmd(:picture_shake, 1, 10, 20, 60)
```

Fait trembler une image (comme les tremblements de l'écran) avec une certaine puissance (nombre de pixels de décalage à gauche et à droite à parcourir selon la vitesse spécifiée dans speed), selon une certaine vitesse (en frames, 1 sec = 60 frames), durant une certaine durée (en frames).

picture_show_enemy(id_picture, id_groupe, position, *x, *y)

```
picture_show_enemy(1, 1, 2)
```

```
cmd(:picture_show_enemy, 1, 1, 2)
```

Affiche (sous forme d'image) un monstre de la base de données en fonction de l'id d'un groupe et de sa position dans le groupe.

Si aucun x et aucun y n'est fourni, la commande utilisera le x et le y spécifié dans la base de données.

Grâce à ce genre de commande (+ celle sur la lecture de la base de données), vous pouvez entreprendre un système de combat via les événements de manière beaucoup plus simple que via les commandes événementielles initiales.

picture_text(id_picture, text, *size=default, *font=default, *c=white, *bold=false, *italic=false, *outline=false, outline_color=black, *shadow=false)

```
picture_text(1, "Saluuuutttt", 10, 10, color(255, 0, 0), true)
```

```
cmd(:picture_text, 1, "Saluuuutttt", 10, 10)
```

Cette commande vous permet d'afficher des lignes de texte sous forme d'image (donc manipulable avec toutes les autres commandes).

Id picture : l'id de l'image sur lequel le texte se trouvera

text : le texte à afficher

x , y : la position du texte

size : la taille (par défaut, 23)

font : n'importe quelle police installée sur le PC (ou dans le répertoire « fonts » du jeu)

color : un objet Color qui correspond à la couleur du texte (vous pouvez utiliser la commande color référencée plus haut dans la description des commandes standards +- p 14).

Bold, italic correspondent au caractère gras et au style, elles prennent soit true soit false.

Outline aussi, présence ou absence d'un contour.

outline_color correspond à un objet color pour la couleur du contour du texte.

Shadow correspond aussi à true ou false pour l'affichage (ou pas) d'un ombrage.

Exemple simple

Un affichage de texte simple :

Commandes de l'évènement

```
@>Script: picture_text(1, "Hello World", 0, 0, 32, "impact", color(0,255,0), true, true, true, color(255,0,0), true)
@>Déplacer une image: n°1, origine: haut-gauche (200,200), zoom: 100%, 100%, opacité 255, Normal, durée: 160 fr
@>Script: picture_shake(1, 2, 5, 60)
@>
```

Un texte généré avec cette commande se comportera de manière identique à une image. Il peut être utile pour générer des dégâts (par exemple).

Exercice pratique : Création d'une jauge

La création d'une jauge est souvent un exercice compliqué. Nous allons voir qu'avec l'Event Extender, c'est très facile.

Pour ce faire, nous avons besoin de 2 images.
Le fond de notre jauge et notre jauge remplie.



Il ne s'agit que d'un fond noir.

Le contenu aura la même taille (ou 2 px de moins en hauteur et en largeur pour les contours par exemple).



Nous allons commencer par afficher nos deux images dans un évènement en processus parallèle qui affichera d'abord le fond puis la jauge remplie.

```
@>Afficher une image: n°1, 'jaugeA', origine: haut-gauche (0,0), zoom: 100%, 100%, opacité: 255, Normal
@>Afficher une image: n°2, 'jaugeB', origine: haut-gauche (0,0), zoom: 100%, 100%, opacité: 255, Normal
@>Boucle
  @>Attendre 1 frames
  @>
: Fin de boucle
```

C'est dans notre boucle que nous modifierons son zoom_x.

En effet, nous savons que le zoom_x prend un pourcentage. Donc on peut admettre que quand l'image 2 aura 100 %, c'est que nos points de vie seront remplis à 100 %.

Donc il nous suffit d'utiliser le calcul de pourcentage sur les hp_courant du héros avec ses hp max.

Ce qui donnerait :

```
picture_zoom_x(2, percent(actor_hp(1), actor_max_hp(1)))
```

Notre image aura donc comme zoom_x le rapport en pourcent du nombre de HP de l'acteur 1 avec son nombre maximum de hp.

Vous pouvez tester, la jauge fonctionne parfaitement.

Voici le code complet de la jauge. En continuant à jouer avec percent et les

changement de tons, vous pourriez faire une jauge qui est verte et devient de plus en plus rouge plus les points de vies descendent !

```
@>Afficher une image: n°1, 'jaugeA', origine: haut-gauche (0,0), zoom: 100
@>Afficher une image: n°2, 'jaugeB', origine: haut-gauche (0,0), zoom: 100
@>Boucle
  @>Attendre 1 frames
  @>Script: picture_zoom_x(2, percent(actor_hp(1), actor_max_hp(1)))
  @>
: Fin de boucle
```

A noter que vous auriez pu rendre le code moins condensé en le séparant ligne par ligne.
Par exemple :

```
@>Afficher une image: n°1, 'jaugeA', origine: haut-gauche (0,0), zoom: 100
@>Afficher une image: n°2, 'jaugeB', origine: haut-gauche (0,0), zoom: 100
@>Boucle
  @>Attendre 1 frames
  @>Script: SV[1] = percent(actor_hp(1), actor_max_hp(1))
:      : picture_zoom_x(2, SV[1])
  @>
: Fin de boucle
@>
```

Dans cet exemple, j'utilise une variable locale pour stocker le pourcentage. C'est un peu inutile mais certains trouvent ça plus logique (ou lisible).

Multi-Panoramas

Nativement, il est possible de manipuler un panorama. Avec l'Event-Extender, il est possible d'en manipuler, en plus de celui par défaut, 20.

Il est possible de modifier beaucoup de leur propriétés, comme la vitesse de défilement, l'auto défilement, le mode de fusion etc.

Les panoramas de l'event-extender n'écrasent pas le panorama « natif » de RPGMaker, ils sont complémentaires.

parallax_show(id, nom, *z=-100, *opacity=255, *auto_x=0, *auto_y=0, *move_x=1, *move_y=1, *blend=0, *zoom_x=100,0, *zoom_y=100,0)

```
parallax_show(1, 'Fond1', -100, 255, 0, 0, 4, 4)
cmd(:parallax_show,1, "Fond1")
```

Affiche un panorama.

Auto_x et auto_y indiquent le défilement automatique horizontal et vertical

move_x et move_y indiquent la vitesse de scrolling du panorama.

0 => le panorama ne bougera pas, 1 il bougera 2 fois moins vite que la carte, 2 il défilera à la même vitesse que la carte, au dessus de 2 il ira plus vite que le défilement de la carte.

parallax_erase(id_parallax)

```
parallax_erase(1)  
cmd(:parallax_erase, 1)
```

Supprime un panorama donné en fonction de son ID.

parallax_z(id_parallax, z)

```
parallax_z(1, 10)  
cmd(:parallax_z, 1, 10)
```

Change la position du panorama sur l'axe Z. Pour rappel, l'axe Z est l'axe qui caractérise la hauteur (la profondeur), plus l'axe Z sera élevé plus il superposera les autres éléments.

parallax_opacity(id_parallax, new_opacity)

```
parallax_opacity(1, 120)  
cmd(:parallax_opacity, 1, 120)
```

Change l'opacité d'un panorama en fonction de son ID.

parallax_autoscroll(id_parallax, horizontal, vertical)

```
parallax_autoscroll(1, 5, 5)  
cmd(:parallax_autoscroll, 1, 0,0)
```

Change les valeur du défilement automatique d'un panorama référencé par son ID.

parallax_scrollspeed(id_parallax, horizontal, vertical)

```
parallax_scrollspeed(1, 5, 5)  
cmd(:parallax_scrollspeed, 1, 0,0)
```

Change les valeur du défilement du scrolling d'un panorama référencé par son ID. (Modifie move_x et move_y).

parallax_zoom_x(id_parallax, new_zoom_x)

```
parallax_zoom_x(1, 120)  
cmd(:parallax_zoom_x, 1, 120)
```

Modifie le zoom_x d'un panorama référencé par son ID.

parallax_zoom_y(id_parallax, new_zoom_y)

```
parallax_zoom_y(1, 120)  
cmd(:parallax_zoom_y, 1, 120)
```

Modifie le zoom_y d'un panorama référencé par son ID.

parallax_zoom(id_parallax, new_zoom_x, *new_zoom_y=-1)

```
parallax_zoom(1, 120)
```

```
cmd(:parallax_zoom, 1, 120, 200)
```

Modifie le zoom d'un panorama en fonction de son ID.

Si une seule valeur de zoom est donnée, elle s'appliquera à zoom_x et zoom_y, par contre, il est possible de spécifier 2 valeurs pour changer le zoom horizontal et le zoom vertical avec des valeurs distinctes.

parallax_blend(id_parallax, new_blend)

```
parallax_blend(1, 1)
```

```
cmd(:parallax_blend, 1, 2)
```

Modifie le mode de fusion d'un panorama référencé par son ID.

(0 = normal, 1 = ajouter, 2 = soustraire. Si un autre nombre est donné, la commande utilisera le mode de fusion normal.)

parallax_tone(id_parallax, rouge, vert, bleu, *gris=0)

```
parallax_tone(1, 255, 10, 10, 20)
```

```
cmd(:parallax_tone, 1, 120, 120, 255)
```

Module de la date et de l'heure

Ce petit module donne des informations sur la date et l'heure. Il peut être très pratique pour créer une horloge ou des comportements définis en fonction de l'heure dans la journée. Les données reprennent les valeurs réelles, la véritable heure pour le joueur et non une heure « en jeu ».

time_year

```
time_year
```

```
cmd(:time_year)
```

Donne l'année (en quatre chiffres).

time_month

```
time_month
```

```
cmd(:time_month)
```

Donne le mois (de 1 à 12).

time_day

```
time_day
```

```
cmd(:time_day)
```

Donne le jour du mois (de 1 à 31).

time_hour

```
time_hour  
cmd(:time_hour)
```

Donne l'heure (de 0 à 23).

time_min

```
time_min  
cmd(:time_min)
```

Donne le nombre de minutes (de 0 à 59).

time_sec

```
time_sec  
cmd(:time_sec)
```

Donne le nombre de secondes (de 0 à 59).

Un exemple simple

C'est un exemple assez minimaliste mais si l'interrupteur 1 n'est pas activé et que l'heure (réelle) est entre 18h et 7h, on change le ton d'image et on active l'interrupteur 1 (pour ne pas répéter le changement de teinte). Si l'interrupteur 1 est activé et que l'heure est comprise entre 7h et 18h, on désactive l'interrupteur et on remet la teinte normale.

Le tout dans un évènement en processus parallèle (un évènement commun par exemple).

```
@>Condition: Script: !S[1] and (time_hour >= 18 or time_hour <= 7)  
  @>Modifier le ton de l'écran: (-68,-68,0,68) sur 60 frames, attendre la fin  
  @>Interrupteur: [0001] = Activé  
  @>  
  : Fin de condition  
@>Condition: Script: S[1] and (time_hour >= 7 or time_hour >= 18)  
  @>Modifier le ton de l'écran: (0,0,0,0) sur 60 frames, attendre la fin  
  @>Interrupteur: [0001] = Désactivé  
  @>  
  : Fin de condition
```

Gestion des sauvegardes

Il est possible de manipuler ses sauvegardes via des commandes (très pratique pour créer son outil de chargement/sauvegarde personnalisé).

save_game(id_save)

```
save_game(1)  
cmd(:save_game, 1)
```

Sauvegarde la partie sur le slot passé en argument.

load_game(id_save, *time=100)

```
load_game(1)
cmd(:load_game, 1, 60)
```

Charge la partie passée en argument avec un temps de transition par défaut à 100 frames.

save_exists?(id_save)

```
save_exists?(1)
cmd(:save_exists?, 1)
```

Renvoie true si une sauvegarde (référéncée par son ID passé en argument) existe, false sinon.

delete_save(id_save)

```
delete_save(1)
cmd(:delete_save, 1)
```

Supprime la sauvegarde via son ID.

import_variable(id_save, id_var)

```
import_variable(1, 12)
cmd(:import_variable, 1, 12)
```

Retourne la valeur de la variable référencée par son ID de la sauvegarde référencée par son ID.

import_switch(id_save, id_switch)

```
import_switch(1, 12)
cmd(:import_switch, 1, 12)
```

Retourne la valeur d'un interrupteur référencé par son ID de la sauvegarde référencée par son ID.

Gestion des zones

Les zones sont un outil qui permet de définir une zone (en fonction de données) et de vérifier si des points y sont inscrits, si la souris les survole, si la souris les clique.

Une zone n'est pas « physique », en effet, elle est purement virtuelle.

Une zone doit être stockée dans une variable (globale ou locale) pour pouvoir l'utiliser.

Comme une zone est virtuelle, vous pouvez choisir l'unité de mesure de votre choix (cases, pixels) pour la créer. Il s'agira d'utiliser les mêmes unités de mesure pour vérifier si des points y sont inclus ou non.

Par exemple, si une zone est définie en cases, il faudra utiliser `player_x` et `player_y` pour savoir si le joueur est inclus dans la zone et non `player_screen_x` et `player_screen_y`.

create_rect_area(x1, y1, x2, y2)

```
V[1] = create_rect_area(0, 0, 15, 15)
```

```
V[1] = cmd(:create_rect_area, 0, 0, 15, 15)
```

Crée une zone rectangulaire.

x1, y1 correspondent aux coordonnées du coin haut gauche. x2, y2 correspondent au coin bas droite.

Il faut stocker la zone dans une variable (locale ou globale) pour pouvoir l'utiliser par après.

create_circle_area(x, y, r)

```
V[1] = create_circle_area(100, 100, 50)
```

```
V[1] = cmd(:create_circle_area, 100, 100, 50)
```

Crée une zone circulaire.

x, y correspondent au point central.

r correspond au rayon du cercle.

create_ellipse_area(x, y, width, height)

```
V[1] = create_ellipse_area(100, 100, 200, 150)
```

```
V[1] = cmd(:create_ellipse_area, 100, 100, 200, 150)
```

Crée une zone en ellipse via des coordonnées (x, y) et une largeur et une hauteur.

create_polygon_area([[x1,y1],[x2,y2]etc...])

```
SV[1] = create_polygon_area([[1,1],[2, 4], [4, 1]])
```

```
V[1] = cmd(:create_polygon_area, [[1,1],[2, 4], [4, 1]])
```

Crée une zone polygonale. Prend en argument une liste de couples de coordonnées. La dernière sera reliée à la première.

Dans cet exemple, la zone sera triangulaire. Vous pouvez donner autant de couple que vous le désirez.

Les zones peuvent être des polygones non convexes, cependant, il s'agit de ne pas abuser car les zones ont leurs limites.

in_area?(area, x, y)

```
in_area?(V[1], 10, 20)
```

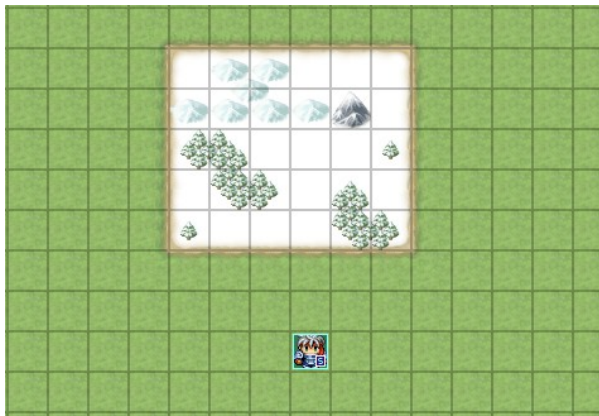
```
cmd(:in_area?, V[1], 10, 20)
```

Vérifie si le point défini par x et y est inclus dans la zone passée en argument (la zone passée en argument correspond à la variable utilisée pour stocker la zone lors de sa création).

(Renvoi true si le point es inscrit, false sinon).

Exemple simple

Voici un petit exemple d'une zone où, quand le héros rentre dedans, il se met à neiger.



Le coin haut-gauche est situé en 5-3 et le coin bas-droit est situé en 10-7. Voici le code dans un évènement en processus parallèle. (Il serait envisageable de l'améliorer en utilisant un interrupteur pour ne changer le climat que dans le cas où il faudrait le changer, donc à l'entrée puis à la sortie de la zone).

```
@>Script: SV[1] = create_rect_area(5, 3, 10, 7)
@>Boucle
  @>Attendre 1 frames
  @>Condition: Script: in_area?(SV[1], player_x, player_y)
    @>Effet météorologique: neige, pendant 0 frames, attendre la fin
  @>
  : Sinon
    @>Effet météorologique: normal, pendant 0 frames, attendre la fin
  @>
  : Fin de condition
@>
: Fin de boucle
@>
```

mouse_hover_area?(area)

```
mouse_hover_area?(V[1])
cmd(:mouse_hover_area?, V[1])
```

Renvoi true si la souris est incluse dans une zone (passée en argument). False sinon.

Cette commande utilise les coordonnées en pixel de la souris.

Si vous avez défini votre zone via des cases, se référer à la commande suivante.

mouse_square_hover_area?(area)

```
mouse_square_hover_area?(V[1])
```

```
cmd(:mouse_square_hover_area?, V[1])
```

Identique à la commande précédente sauf qu'elle fonctionne dans le cas où une zone a été définie en cases (et non en pixels).

Ces deux commandes sont identiques, sauf qu'elles changent le système métrique de l'évaluation d'un point dans une zone.

mouse_clicked_area?(area, button)

```
mouse_clicked_area?(V[1],:mouse_left)
```

```
cmd(:mouse_clicked_area?, V[1],:mouse_center)
```

Renvoie true si la souris est incluse et clique dans une zone (passée en argument). False sinon.

Cette commande utilise les coordonnées en pixels de la souris.

Si vous avez défini votre zone via des cases, se référer à la commande qui contient square dans son nom.

mouse_triggered_area?(area, button)

```
mouse_triggered_area?(V[1],:mouse_left)
```

```
cmd(:mouse_triggered_area?, V[1],:mouse_center)
```

Renvoie true si la souris est incluse et clique (ponctuellement) dans une zone (passée en argument).

mouse_repeated_area?(area, button)

```
mouse_repeated_area?(V[1],:mouse_left)
```

```
cmd(:mouse_repeated_area?, V[1],:mouse_center)
```

Renvoie true si la souris est incluse et clique (de manière répétée) dans une zone (passée en argument).

mouse_released_area?(area, button)

```
mouse_released_area?(V[1],:mouse_left)
```

```
cmd(:mouse_released_area?, V[1],:mouse_center)
```

Renvoie true si la souris est relâchée dans une zone (passée en argument).

mouse_square_clicked_area?(area, button)

```
mouse_square_clicked_area?(V[1],:mouse_left)
```

```
cmd(:mouse_square_clicked_area?, V[1],:mouse_center)
```

Identique que la commande précédente sauf qu'elle fonctionne dans le cas où une zone a été définie en cases (et non en pixel).

mouse_square_triggered_area?(area, button)

```
mouse_square_triggered_area?(V[1],:mouse_left)
```

```
cmd(:mouse_square_triggered_area?, V[1],:mouse_center)
```

Identique que la commande précédente sauf qu'elle fonctionne dans le cas où une zone a été définie en cases (et non en pixel).

mouse_square_repeated_area?(area, button)

```
mouse_square_repeated_area?(V[1],:mouse_left)  
cmd(:mouse_square_repeated_area?, V[1],:mouse_center)
```

Identique que la commande précédente sauf qu'elle fonctionne dans le cas où une zone a été définie en cases (et non en pixel).

mouse_square_released_area?(area, button)

```
mouse_square_released_area?(V[1],:mouse_left)  
cmd(:mouse_square_released_area?, V[1],:mouse_center)
```

Identique que la commande précédente sauf qu'elle fonctionne dans le cas où une zone a été définie en cases (et non en pixel).

Zone de texte saisissable au clavier

Un problème récurrent en Event-making est la création de champs de textes modifiables.

Avec l'event-extend, il est possible de faire des zones de texte modifiables au clavier.

Ces zones de textes peuvent recevoir du texte, des entiers et des nombres à virgules.

Leur valeur est récupérable et ces champs sont activables.

Lorsqu'un champ est activé, tous les autres champs sont désactivés.

Il est possible de détecter le survol ou le clique d'une zone de texte modifiable.

Il est aussi possible de leur définir un nombre maximum de caractères (dans le cas où la zone est textuelle) et une intervalle de nombres dans le cas où la zone est numérique (entier ou nombres à virgule). Par défaut, le « look » des fenêtres de saisie admet le Windowskin défini dans la base de données, cependant, il est possible de changer sa teinte.

create_textfield(type, x, y, width, *text=vide, *align=:left, *range=-1, *height=38)

```
V[1] = create_textfield(:text, 10, 10, 200, "votre nom")  
V[1] = cmd(:create_textfield,:float, 10, 10, 200)
```

Crée un champ de texte (la commande « retourne une zone de texte », il faut donc la sauvegarder dans une variable globale ou locale).

- Type : peut prendre 3 valeurs =>:text pour une zone de texte, :integer pour un entier et :float pour un nombre à virgule.
- x, y : coordonnées de position de la zone de texte
- width : la largeur de la zone de texte
- text: la première valeur à afficher dans la zone (par défaut ce champ est vide)
- align : peut prendre trois valeurs : pour un alignement du texte à

- gauche :left, pour un alignement du texte au centre :center et :right pour aligner le texte à droite (par défaut le texte est aligné à gauche)
- range : Dans le cas d'une zone texte, il s'agit d'un entier qui limite le nombre de caractères. Pour les champs numériques (:float et :integer) il est possible de passer un intervalle par exemple (0..99) permettra au champs d'aller de 0 à 99, (-999..999) permettra au champs d'aller de -999 à 999. Pour ne mettre aucune limite il suffit de passer -1. Par défaut ce champ est à -1.
- height :La hauteur du champ texte (par défaut elle vaut 38).

Par défaut, lorsqu'une fenêtre est créée, elle est inactive. Nous verrons plus loin comment il est possible d'activer une zone de texte modifiable.

Sans être active, une zone de texte est juste une fenêtre qui peut afficher une ligne de texte. (Cependant, lorsqu'il faut afficher du texte, je recommande la commande `picture_text` qui est prévue pour ça).

Si la commande `create_textfield` doit être liée à une variable, c'est parce que c'est avec cette variable que l'on pourra travailler sur notre zone de texte.

erase_textfield(textfield)

```
erase_textfield(V[1])
cmd(:erase_textfield, V[1])
```

Supprime la zone de texte passée en argument.

get_textfield_value(textfield)

```
get_textfield_value(V[1])
cmd(:get_textfield_value, V[1])
```

Renvoie la valeur contenue dans la zone de texte.

set_textfield_value(textfield, text)

```
set_textfield_value(V[1], "Nouveau Texte")
cmd(:set_textfield_value, V[1], "Un texte")
```

Change la valeur de la zone de texte passée en argument.

activate_textfield(textfield)

```
activate_textfield(V[1])
cmd(:activate_textfield, V[1])
```

Active la zone de texte passée en argument.

Il devient possible de changer son contenu au clavier.

Lorsqu'une zone est activée, elle désactive toutes les autres.

unactivate_textfield(textfield)

```
unactivate_textfield(V[1])
cmd(:unactivate_textfield, V[1])
```

Désactive la zone de texte passée en argument. (Une zone désactivée est toujours visible, mais elle ne réagit plus aux saisies clavier).

textfield_activated?(textfield)

```
textfield_activated?(V[1])  
cmd(:textfield_activated?, V[1])
```

Renvoie true si le champ de texte passé en argument est activé, false sinon.

unactivate_all_textfield

```
unactivate_textfield(V[1])  
cmd(:unactivate_textfield, V[1])
```

Désactive toutes les zones de textes. (Une zone désactivée est toujours visible, mais elle ne réagit plus aux saisies clavier).

textfield_hover?(textfield)

```
textfield_hover?(V[1])  
cmd(:textfield_hover?, V[1])
```

Renvoie true si la souris est sur le champ de texte passé en argument, false sinon.

textfield_clicked?(textfield, button)

```
textfield_clicked?(V[1], :mouse_left)  
cmd(:textfield_clicked?, V[1], :mouse_left)
```

Renvoie true si la souris est sur le champ de texte passé en argument et qu'elle est cliquée par le bouton passé en argument, false sinon.

textfield_opacity(textfield, opacity)

```
textfield_opacity(V[1], 200)  
cmd(:textfield_opacity, 200)
```

Change l'opacité de la zone de texte passée en argument (de 0 à 255).

textfield_tone(textfield, rouge, vert, bleu, *gris=0)

```
textfield_tone(V[1], 255, 10, 10, 20)  
cmd(:textfield_tone, V[1], 120, 120, 255)
```

Change la teinte d'un champ de texte passé en argument avec des valeurs de rouge, vert, bleu allant de -255 à 255, et d'une teinte grise qui par défaut vaut 0 et peut aller de 0 à 255.

Exemple simple

Voici un exemple d'utilisation. Nous allons écrire en event-making, un petit système pour permettre au héros de changer son nom.

Nous allons donc créer un évènement qui se déclenchera via la touche action. Il créera une zone de texte avec comme valeur le nom actuel du personnage. Lancera une boucle dans lequel il attendra l'appui de la touche « enter » pour valider la saisie et appliquera le nom passé en dans la zone de texte au héros 1.

Premièrement, nous allons initialiser notre champ de texte. Nous l'afficherons au dessus de l'évènement qui déclenche le système.

```
@>Commentaire: Création de la zone de texte
@>Script:
:      : x = event_screen_x(@event_id) - 100
:      : y = event_screen_y(@event_id) - 64
:      : SV[1] = create_textfield(:text, x, y, 200, actor_name(1), :center)
```

Comme ma zone fait 200 pixels de largeur, je la décale de 100 pixels pour qu'elle soit centrée sur mon évènement.
J'ai utilisé des variables x et y et non des variables type V et SV parce que je n'en aurai besoin que maintenant.
Vous pouvez tester le système. Vous devriez avoir quelque chose comme ça lorsque vous activez l'évènement :



Il faut maintenant activer notre zone de texte pour lui permettre de détecter la saisie du clavier :

```
@>Commentaire: Création de la zone de texte
@>Script:
:      : x = event_screen_x(@event_id) - 100
:      : y = event_screen_y(@event_id) - 64
:      : SV[1] = create_textfield(:text, x, y, 200, actor_name(1), :center)
@>Commentaire: Activation de la zone de texte
@>Script: activate_textfield(SV[1])
```

Si vous testez notre système maintenant, vous verrez qu'il est possible de changer le contenu de la zone de texte.
Mais si l'on utilise les chiffres, le héros bouge. Il faudra donc utiliser une boucle qui détectera si la touche enter est entrée pour valider le nouveau nom du héros. Ce qui nous donne :

```

@>Afficher un message: Aucun portrait, Normal, Bas
:           : Salut, peux-tu me dire ton nom?
@>Commentaire: Création de la zone de texte
@>Script:
:           : x = event_screen_x(@event_id) - 100
:           : y = event_screen_y(@event_id) - 64
:           : SV[1] = create_textfield(:text, x, y, 200, actor_name(1), :center)
@>Commentaire: Activation de la zone de texte
@>Script: activate_textfield(SV[1])
@>Commentaire: Boucle de détection de la pression de la touche [Enter]
@>Boucle
  @>Attendre 1 frames
  @>Condition: Script: key_trigger?(:enter)
    @>Sortir de la boucle
    @>
  : Fin de condition
@>
: Fin de boucle

```

Maintenant que l'on a notre boucle, à sa sortie il faudra extraire le résultat de la zone de texte, l'attribuer comme nom à notre héros 1 et ensuite supprimer la zone de texte (et afficher un petit message:)).

```

@>Afficher un message: Aucun portrait, Normal, Bas
:           : Salut, peux-tu me dire ton nom?
@>Commentaire: Création de la zone de texte
@>Script:
:           : x = event_screen_x(@event_id) - 100
:           : y = event_screen_y(@event_id) - 64
:           : SV[1] = create_textfield(:text, x, y, 200, actor_name(1), :center)
@>Commentaire: Activation de la zone de texte
@>Script: activate_textfield(SV[1])
@>Commentaire: Boucle de détection de la pression de la touche [Enter]
@>Boucle
  @>Attendre 1 frames
  @>Condition: Script: key_trigger?(:enter)
    @>Sortir de la boucle
    @>
  : Fin de condition
@>
: Fin de boucle
@>Commentaire: Finalisation du système
@>Script: nom = get_textfield_value(SV[1])
:           : erase_textfield(SV[1])
:           : set_actor_name(1, nom)
@>Afficher un message: Aucun portrait, Normal, Bas
:           : Oh, tu t'appelles \N[1], c'est un joli prénom !

```

Nous avons donc un outil, plus original que l'original, pour changer le prénom du héros (via le clavier). (Et dans un nombre de lignes assez réduit !)
 A noter qu'il aurait été possible, plutôt que de créer une variable nom, de faire, avant la suppression de la zone de texte :
 set_actor_name(1, get_textfield_value(SV[1])) ce qui aurait économisé une ligne...

Interface réseau (Sockets)

Comme dans l'ancienne version, il est possible de se connecter à un serveur via le protocole TCP/IP et d'envoyer/recevoir des messages.

Cette collection de commande permet d'ajouter une dimension multi joueur à votre projet mais dans la limite de l'admissible. En effet, il ne serait pas possible de réaliser un MMORPG (le moteur de RPG Maker ne le supporterait pas) mais par contre, il est tout à fait envisageable de concevoir des jeux collaboratifs à plusieurs, des scores en lignes, des messageries entre les joueurs, des échanges de données.

Cette série de commande est exclusivement réservée à des gens expérimentés et l'Event Extender ne fournit « que » le client mais il faudra un serveur (application) pour traiter les données envoyées par les clients.

Chacune de ces commandes possède une version « single » qui évite de devoir relayer le socket en argument. Les version « single » ne permettent pas de faire des connexions multiples, les commandes classiques le permettent.

server_connect(host, port)

```
V[1] = server_connect("127.0.0.1", 9999)
V[1] = cmd(:server_connect, "127.0.0.1", 9999)
```

Cette commande prend en argument une adresse IP (sous forme de chaîne de caractères) et un port (sous forme d'entier).

Cette commande renvoie le socket sur laquelle la connexion est effectuée.

C'est ce socket qu'il faudra utiliser pour les fonctions de serveur qui requièrent un socket en argument. Il faut donc lier le résultat à une variable.

server_single_connect(host, port)

```
server_single_connect("127.0.0.1", 9999)
cmd(:server_single_connect, "127.0.0.1", 9999)
```

Cette commande est identique à la précédente sauf qu'elle ne retourne pas de socket. En effet, elle n'admet qu'une seule connexion et toutes les autres commandes relative à l'interface réseau « single » utiliseront le socket généré par cette commande.

server_send(socket, message)

```
server_send(V[1], "Coucou le serveur !")
cmd(:server_send, V[1], "Coucou le serveur !")
```

Envoie un message au serveur passé en argument (via la variable utilisée pour lier le socket dans la commande server_connect).

server_single_send(message)

```
server_single_send("Coucou le serveur !")
cmd(:server_single_send, "Coucou le serveur !")
```

Envoie un message au serveur lorsqu'une connexion à été lancée avec `server_single_connect`. Il ne faut donc pas spécifier son socket en argument.

`server_recv(socket, *taille_max=256)`

```
V[2] = server_recv(V[1])  
V[2] = cmd(:server_recv, V[1], 120)
```

Reçoit un message du serveur passé en argument. Il faut boucler sur sa réception. Il est possible de paramétrer une taille de réception max. Par défaut, cette taille correspond à 256.

Tant que la commande ne reçoit rien, elle renvoie « false ».

`server_single_recv(*taille_max=256)`

```
V[2] = server_single_recv  
V[2] = cmd(:server_single_recv, 120)
```

Fonctionne identiquement à la commande précédente mis à part qu'il ne faut pas spécifier le serveur en argument car elle utilise celle de `server_single_connection`.

`server_wait_recv(socket, *taille_max=256)`

```
V[2] = server_wait_recv(V[1])  
V[2] = cmd(:server_wait_recv, V[1], 120)
```

Identique à la commande `server_wait` sauf qu'elle intègre la logique d'attente d'un message, la commande attend un message avant de renvoyer une valeur. C'est celle qui est le plus commun à utiliser en Event-Making.

`server_single_wait_recv(*taille_max=256)`

```
V[2] = server_single_wait_recv  
V[2] = cmd(:server_single_wait_recv, 120)
```

Identique à la commande précédente sauf qu'elle utilise le serveur de `single_connect` comme serveur et donc il ne faut pas lui passer en argument.

`server_close_connection(socket)`

```
server_close_connection(V[1])  
cmd(:server_close_connection, V[1])
```

Ferme la connexion avec le serveur passé en argument.

`server_single_close_connection`

```
server_single_close_connection  
cmd(:server_single_close_connection)
```

Ferme la connexion « single » générée avec `server_single_connect`.

Commandes diverses

Nous allons voir les commandes diverses et variées mais pas les moindres !

include_page(map_id, event_id, page_id)

```
include_page(1, 2, 1)
```

```
cmd(:include_page, 3, 2, 1)
```

Inclus une page (page_id) d'un évènement (event_id) d'une map (map_id) dans un autre évènement.

L'avantage de cette commande est quelle permet de modulariser le code. De stocker, dans un autre évènement, du code à réutiliser.

invoke_event(map_id, event_id, new_id, *x=nil, *y=nil)

```
invoke_event(8,2,10,1,1)
```

```
cmd(:invoke_event, 8, 2, 10)
```

Place un évènement (référéncé par son ID), d'une map (référéncée par son ID) sur la carte avec un autre id défini par new_id. Si aucun X et aucun Y ne sont fournis, il se placera à la même position que sur la map d'où il est importé.

Cette commande est identique à la commande « placer évènement » sauf qu'elle permet de placer des évènements de maps différentes. Attention de ne pas écraser l'ID d'un autre évènement. Des conflits d'ID's sont possibles si les évènements sont importés sans vérifier qu'ils se placent à des ID's libres.

windows_username

```
windows_username
```

```
cmd(:windows_username)
```

Retourne (sous forme de chaîne de caractères) le nom de la session Windows. Dans l'exemple fourni dans la manipulation des zones de textes, à la place de actor_name(1) comme texte par défaut, il aurait été amusant de lui passer windows_username comme argument pour que par défaut le personnage soit nommé par le nom de la session windows. Sur mon ordinateur, ça aurait donné « Pierrot ».

angle_xy(source_x, source_y, cible_x, cible_y)

```
angle_xy(0,0, 100, 100)
```

```
cmd(:angle_xy, 0, 0, 100, 100)
```

Calcule l'angle par rapport à l'axe horizontal entre une source et une cible référencée par leurs X et leurs Y.

Exemple simple

Nous allons nous servir de la commande angle_xy pour se servir de la souris comme direction d'une barre verticale.

Pour cela je me sers d'une image orientée par son coin haut-gauche, (j'ai

personnellement utilisé une barre) et placée au centre de l'écran, donc en 272x208. Voici comment je change l'angle de l'image en fonction de l'angle défini par le curseur :

```
@>Afficher une image: n°1, 'minarrow', origine: haut-gauche (272,208), zoom: 100%, 100%, opacité: 255, Normal
@>Boucle
  @>Attendre 1 frames
  @>Script: picture_angle(1, angle_xy(272, 208, mouse_x, mouse_y))
  @>
: Fin de boucle
@>
```

rtp_path

```
rtp_path
cmd(:rtp_path)
```

Retourne, sous forme de chaîne de caractères, le chemin où est installé le RTP. C'est principalement un outil pour les scripteurs.

Conclusion des commandes

Nous en avons fini avec la description des commandes implémentées par défaut dans l'Event Extender. Ce script a été pensé pour être extensible, il est donc tout à fait admissible que des scripteurs raffinent le module Command pour lui greffer plus de fonctionnalités. Pour ça il suffit d'étendre le module Command et d'y ajouter le code des commandes désirées.

J'invite les scripteurs désireux d'apporter des plugins au script de lire le code source général de l'Event Extender pour profiter des utilitaires déjà implémentés.

Étendre le module Command

Voici un petit exemple d'ajout de commande. À ajouter en dessous de l'Event Extender.

```
#=====
# ** Command
#-----
# Adds easily usable commands
#=====
module Command
  # Ici vous pouvez ajouter vos commandes
  # Par exemple, une commande qui affiche "bonjour"
  def bonjour
    msgbox("Bonjour !")
  end
end

# La commande cmd(:bonjour) ou bonjour est maintenant ajoutée
```

N'hésitez donc pas à ajouter vos propres commandes dans le script.

Merci !

Les outils complémentaires

L'outil principal de l'Event Extender est son module Command, cependant, il existe quelques outils complémentaires que je vais documenter ici. Ils ont été pensés pour faciliter votre Event Making.

Le testeur de teintes

Il est souvent très difficile d'utiliser le modificateur de teinte de l'écran des commandes événementielles car il n'affiche pas de rendu « in game » de la teinte choisie.

Personnellement, il m'arrivait de passer de longs moments à tenter de le paramétrer en testant sans arrêt mon jeu pour être sûr d'avoir une teinte qui me plaisait.

Le testeur de teinte est un outil qui permet de tester les teintes en jeu. Pour cela il suffit d'appuyer sur la touche F3 du clavier lorsque votre joueur est sur une carte.

Une fenêtre de modification fera son apparition. Pour la fermer, il suffit de ré-appuyer sur la touche F3.

Utilisation du testeur de teintes

Lorsque vous avez appuyé sur la touche F3, vous devriez voir ceci apparaître à l'écran (rassurez-vous, cette commande n'est valide que quand elle est lancée depuis une partie lancée depuis l'éditeur).



En appuyant sur F3 vous pourrez le faire disparaître. Vous pouvez changer la teinte en modifiant les barres de défilements (et en faisant glisser le bouton correspondant à sa couleur).

En cliquant sur un des champs de texte vous pouvez l'activer et changer sa

valeur au clavier.

Le bouton try Tone fera une simulation de la transition du changement de teinte avec la teinte initiale et la teinte générée avec le testeur de teinte. Le champ Time peut être modifié pour augmenter la durée de la transition. La case « Wait ? » permet de dire si le jeu doit attendre la fin de la transition (si cochée) ou pas.

Le bouton « Make Command » génère la commande dans le presse-papier. Une fois que vous avez appuyé sur ce bouton, vous pouvez fermer le jeu et vous rendre dans un événement et utiliser clique droit/coller pour coller la commande générée avec le testeur de teinte dans un événement.

Un message vous avertira que la commande a bien été collée dans le presse-papier.

Modifier la touche de lancement du testeur de teintes

Il est possible de modifier la touche qui lance le testeur de teintes en vous rendant dans le script Event Extender, tout au début, il y a un module « Configuration » aux alentours de la ligne 37. Il suffit de changer la valeur de `KEY_TONE_MANAGER` qui par défaut vaut « :f3 » par une des touches proposées dans le module Keyboard.

Le testeur de script

Il est aussi courant de vouloir tester des petits scripts (ou des commandes). Pour cela il existe un testeur de code in Game. Qui permet de tester des commandes, de modifier des variables directement in Game.

Utiliser le testeur de script

L'objectif de cet outil est de rendre facile le test de commande, de modification de variables etc.

Vous pouvez lancer le testeur de script en appuyant sur F4 (et F4 pour le masquer).



Il est maintenant possible d'écrire du script et de le tester en appuyant sur [Enter]. Pour afficher un message vous pouvez soit vous servir de la console en écrivant « p windows_username » ou alors utiliser message box : « msgbox(windows_username) » (dans cet exemple j'afficherai le nom de la session windows d'abord dans la console ensuite dans une fenêtre « pop up »).

Vous pouvez aussi tester des commandes de l'Event Extender comme par exemple « buzz 0 » etc.

Vous pouvez aussi modifier des variables (via V, SV), les switches (via S, SS) etc. C'est un outil de test très pratique. Le bouton « Make Command » est identique à celui du testeur de teinte, il va placer l'appel de script dans le presse-papier, et il sera possible de le coller dans un événement.

Voici un petit exemple d'utilisation :



Modifier la touche de lancement du testeur de scripts

Il est possible de modifier la touche qui lance le testeur de scripts en vous rendant dans le script Event Extender, tout au début, il y a un module « Configuration » aux alentours de la ligne 37. Il suffit de changer la valeur de `KEY_INGAME_EVAL` qui par défaut vaut « :f4 » par une des touches proposées dans le module Keyboard.

La base de données alternative

La base de données de RPG Maker est assez complète mais, elle souffre d'un gros problème, elle est totalement immuable.

On ne peut pas altérer sa structure, ce qui fait que pour étendre les données de la base de données, il faut ruser et utiliser les commentaires (notes) de chaque champs de la base de données.

Pour éviter ça, je me suis inspiré du script de Avygeil ; Database Management. L'objectif est d'offrir un moyen de structurer une base de données où les structures de données seraient libres.

Une table

Une table est une représentation de données. Par exemple ; les armes de la base de données sont une table, les monstres en sont une autre, les groupes de monstres en sont aussi une.

Dans l'Event extender, il est possible de créer ses propres tables. Pour cela, en

dessous de l'Event extender (dans un nouvel emplacement script par exemple) nous allons créer des tables.

Pour cela il faut utiliser la fonction « `create_table(titre, *champs)` ». La représentation des champs est un petit peu particulière car elle doit définir un nom de champ et un type de champ.

La liste des types possible est :

- `:int` ou `:interger` => représente un nombre entier
- `:float` => représente un nombre à virgule
- `:bool` ou `:boolean` ou `:switch` => une variable qui peut valoir true ou false (un interrupteur)
- `:text` ou `:string` => du texte.

Voici un exemple de création de table :

```
create_table("dragons", id: :int, nom: :string, age: :int, male: :boolean)
```

Ici nous avons créé une table dragons qui peut être représenté sous cette forme :

dragons	
id	int
nom	string
age	int
male	boolean

Il est donc possible de représenter des dragons via ces caractéristiques. Le nombre de champs est à priori illimité (enfin, je doute qu'il soit humainement viable de gérer plus de 30 champs mais bon...).

Nous pouvons nous servir des tables pour représenter toute formes de données, des objets spéciaux, des animaux, des monstres etc.

Remplir les tables

Les tables sont comme les tiroirs d'une armoire. Chacun de ces tiroirs étant ordonné par les champs typés que nous avons défini. Il est donc possible de remplir notre armoire de données.

Nous allons voir comment remplir nos tables de données statiques pour représenter des informations.

Il faut récupérer la table. Pour ça il suffit d'utiliser `T["nom de la table"]` ou `Tables["nom de la table"]`. Ensuite nous allons la remplir en utilisant l'opérateur `<< [liste de données]`.

Par exemple voici un schéma qui remplit notre table de dragons :

```
create_table("dragons", id: :int, nom: :string, age: :int, male: :boolean)
```

```
Tables["dragons"] << [0, "dragon blanc", 50, true]
Tables["dragons"] << [1, "dragon rouge", 150, true]
Tables["dragons"] << [2, "dragon bleu", 250, false]
```

Nous avons donc créé une table dragons et nous l'avons remplie de 3 dragons. Il est possible de créer autant de tables que l'on veut et des les remplir du nombre d'entrées que l'on désire.

J'ai l'habitude de créer un champ « id » dans toutes mes tables pour accéder rapidement à des entrées via leurs ID's (nous verrons ça à la rubrique suivante pour interroger des tables).

Interroger la base de données

Maintenant que nous savons comment créer des tables et remplir les tables de notre base de données alternative, nous allons voir comment interroger nos tables.

Size

```
Tables["dragons"].size
T["dragons"].size
```

Renvoie le nombre d'entrées dans une table.

Count

```
Tables["dragons"].count{|dragon|dragon.id == 0}
T["dragons"].count{|dragon|dragon.age == 150 or dragon.id == 0}
```

Compte le nombre d'entrées qui correspond au prédicat passé en argument.

La forme d'un prédicat est comme ceci :

{|nom_libre| liste de conditions}

Le nom entre les | est libre, personnellement j'utilise le nom de la table au singulier. Les conditions peuvent être liées par des opérateurs logiques (se référer à la documentation sur les switches pour plus de précisions).

Dans les deux exemples données, le premier renverrait 1, il n'y a qu'un seul dragon dont l'id vaut 0.

Le second exemple renverrait 2 car il existe 1 dragon âgé de 150 ans et 1 dragon dont l'id vaut 0. (Comme la condition s'interroge sur un dragon dont l'age vaut 150 OU l'id vaut 0, il prend les deux qui respectent cette condition).

Find

```
V[1] = Tables["dragons"].find{|dragon|dragon.id == 0}
V[1] = T["dragons"].find{|dragon|dragon.age == 150 or dragon.id == 0}
```

Récupère la première entrée qui correspond au prédicat passé en argument.

La forme d'un prédicat est comme ceci :

{|nom_libre| liste de conditions}

Le nom entre les | est libre, personnellement j'utilise le nom de la table au

singulier. Les conditions peuvent être liées par des opérateurs logiques (se référer à la documentation sur les switches pour plus de précisions). Comme dans l'exemple j'ai lié le résultat à la variable 1, je pourrais accéder aux propriétés du dragon en faisant V[1].age ou V[1].nom etc. (en utilisant les champs définis dans le mapping de la base de données).

Select

```
V[1] = Tables["dragons"].select{|dragon|dragon.male}
```

```
V[1] = T["dragons"].find{|dragon|dragon.age > 150 or dragon.id > 0}
```

Récupère toutes les entrées qui correspondent au prédicat passé en argument.

La forme d'un prédicat est comme ceci :

{|nom_libre| liste de conditions}

Le nom entre les | est libre, personnellement j'utilise le nom de la table au singulier. Les conditions peuvent être liées par des opérateurs logiques (se référer à la documentation sur les switches pour plus de précisions).

Comme dans l'exemple j'ai lié le résultat à la variable 1, je pourrais accéder aux propriétés du dragon en faisant V[1].age ou V[1].nom etc. (en utilisant les champs définis dans le mapping de la base de données).

Le résultat est une liste donc il faudra utiliser sur la variable liée au résultat (dans notre exemple V[1]) la méthode length, soit V[1].length pour savoir le nombre de résultats retourné par la commande. Et pour accéder à chaque élément de la table utiliser V[1][0] pour le premier, V[1][1] pour le deuxième, V[1][2] pour le troisième et on peut donc accéder aux propriétés de chaque dragon via V[1][0].age par exemple.

Le résultat va de 0 à la taille du résultat-1, et si aucun résultat ne correspond au prédicat, V[1] vaudra [] et sa taille vaudra 0.

Conclusion sur la base de données

Ce module peut paraître un peu complexe à utiliser pour les néophytes, cependant, je vous invite à tâcher de vous familiariser avec ce dernier. En effet, il permet de réellement casser certaines limites de l'Event Making en vous permettant de mapper vous même votre propre base de données !

L'éditeur de commandes

Cette partie n'a pas été relue par Ulis (car elle a été rédigée après), désolé si je manque de clarté.

L'apprentissage des commandes est une étape complexe car il y en a énormément, c'est donc pour ça que Nuki, avec la complicité de Hiino (et un tout petit peu de moi) (et une fois de plus l'aide de Zeus) nous avons créé un petit éditeur pour générer ses commandes EventExtender.

Son objectif est d'offrir un outil qui peut à la fois faire office d'aide mémoire mais aussi de générateur de commandes facilement copiables/collables. Cet utilitaire a été prévu pour s'adapter à l'Event Extender de l'utilisateur et il est prit en

charge à partir de la version 4.5 de l'Event Extender. Dans cette section nous verrons comment installer/utiliser l'Event Extender Editor mais aussi comment préparer ses commandes « personnalisées » à rentrer dans l'éditeur sans devoir tout le temps changer d'éditeur.

Mise en garde

Même si l'éditeur offre un outil relativement confortable, une lecture de la documentation est tout de même conseillée pour comprendre, plus en détail, le fonctionnement de chaque commandes.

L'éditeur lit le projet auquel il est lié, il faut donc impérativement comme la boucle de lancement de votre jeu, soit le script généralement appelé « Main » soit appelé « Main », sinon l'éditeur peut ne jamais se lancer.

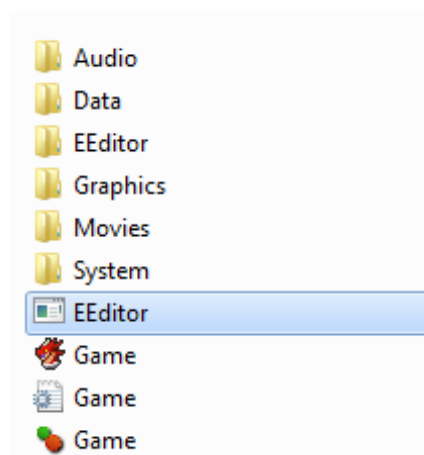
Téléchargement

Lien de téléchargement :

1. <http://biloucorp.com/BCW/Grim/EEEditor.zip>

Installation

Il vous suffit de décompresser le fichier *EEEditor.zip* dans le répertoire du projet qui utilise l'Event Extender, vous pouvez ensuite supprimer le fichier .ZIP. Votre répertoire devrait ressembler à ceci :

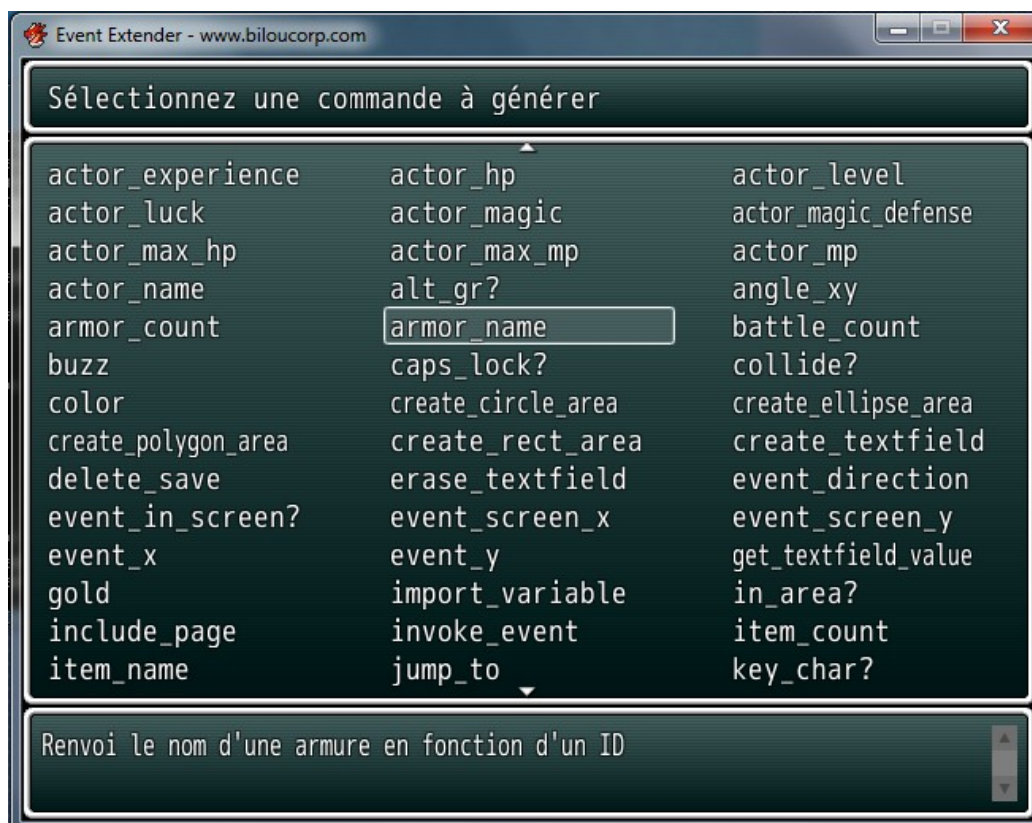


Pour lancer l'éditeur, il suffit de cliquer sur Eeditor.exe et une fenêtre noire s'affichera (elle indiquera « Loading Event Extender Editor ») le temps que l'éditeur se charge. Si cette fenêtre noire (qui disparaît une fois l'éditeur lancé) vous dérange, vous pouvez lancer l'éditeur depuis le répertoire Eeditor, Game.exe. Comme l'éditeur est un projet RPG Maker VX Ace, il est possible qu'il mette quelques secondes à se lancer.

Liste des commandes

Le premier écran que vous devriez voir est celui qui affiche la liste des commandes Event Extender. C'est sur cet écran que vous choisirez la

commande à créer :



Cette page affiche l'intégralité des commandes « correctement » implémentées dans l'Event Extender situé dans le projet parent.

Il est possible de se déplacer au moyen des flèches directionnelles et d'utiliser la souris pour faire défiler l'espace qui donne la description de la commande (en bas) lorsque la description est « trop grande » pour être affichée dans le cadre.

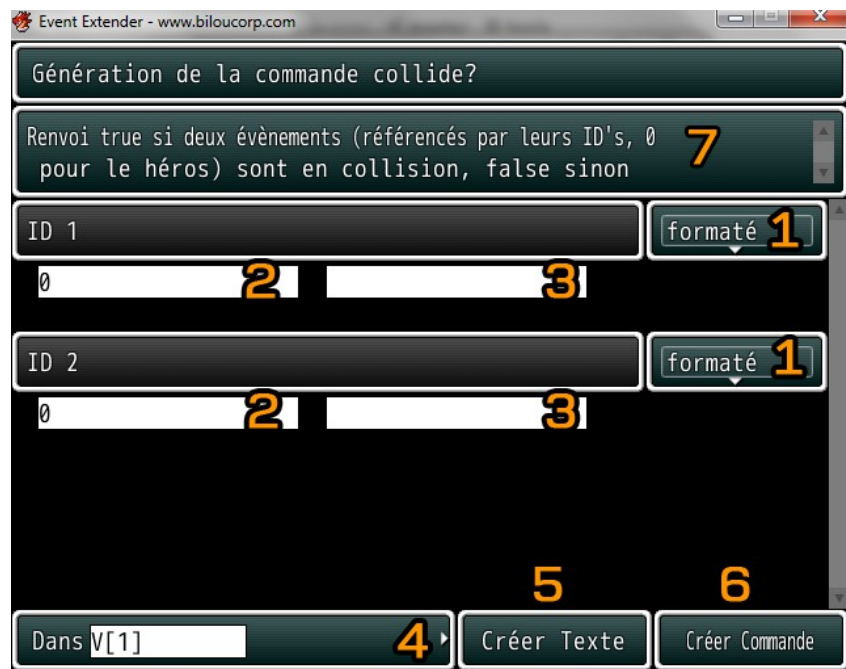
Appuyer sur Enter enverra vers une autre Scène pour « construire » la commande choisie.

Créer une commande particulière

En sélectionnant une commande, c'est une autre scène qui se lance. Notez qu'il est possible de revenir à la fenêtre de sélection de commande en appuyant simplement sur [ESC] ou [F12].

Chaque élément « modifiable » doit être cliqué pour être activé.

Voici le schéma de cette fenêtre.



Event Extender - www.biloucorp.com

Génération de la commande collide?

Renvoi true si deux évènements (référéncés par leurs ID's, 0 pour le héros) sont en collision, false sinon 7

ID 1 formaté 1
0 2 3

ID 2 formaté 1
0 2 3

Dans V[1] 4

Créer Texte 5

Créer Commande 6

1. Ces boutons sont cliquables, ils permettent de définir quel champ de texte pour les arguments seront choisis (voir 2 et 3), le « formaté » valide le champ de texte formaté pour recevoir les bons arguments au bon format, le champ libre valide un champ qui peut prendre n'importe quel format, par exemple, une autre commande. Il faut, quand on génère la commande, valider lequel des deux champs sera validé par argument de la commande.
2. Il s'agit du champ de texte « formaté », par exemple, si la commande attend un nombre, par exemple un ID d'évènement, elle n'acceptera que des chiffres.
3. Il s'agit du champ « libre » qui peut prendre une commande ou une variable (par exemple SV[1] ou V[3] ou encore mouse_x). Elle est placée dans la génération de commande si son module est défini en « libre » au lieu de « formaté ».
4. Dans le cas où la fonction renvoie quelque chose (un message, une valeur) il est possible de sélectionner directement dans quoi stocker ce message. On peut remplacer le champ par tout type de variable (par défaut il s'agit de la variable globale 1).
5. Permet de sauvegarder le texte à coller dans l'appel de script.
6. Permet de générer la commande à coller dans un évènement.
7. Description succincte de la commande.

Conclusion finale

Voici la seconde draft de la documentation, elle devrait être assez claire pour réaliser pleins de petites choses sympa ! Si vous avez des questions n'hésitez pas à m'ajouter sur MSN (pierre.ruyter____AROBAS____hotmail.fr) ou de m'envoyer des MP.

J'espère que ce script vous amusera et je vous invite à visiter mon blog (cf en-tête), j'écris peu de choses mais quand j'en écris... j'en écris... A bientôt !

J'espère de tout cœur que ce script vous sera utile !

Grim

Conditions d'utilisations

Ce script est entièrement libre, aucune créditation obligatoire, peut être utilisé dans un projet commercial sans aucune limite. Le script peut être partagé partout sans limite. L'auteur ne doit pas être respécifié. Bonne utilisation.