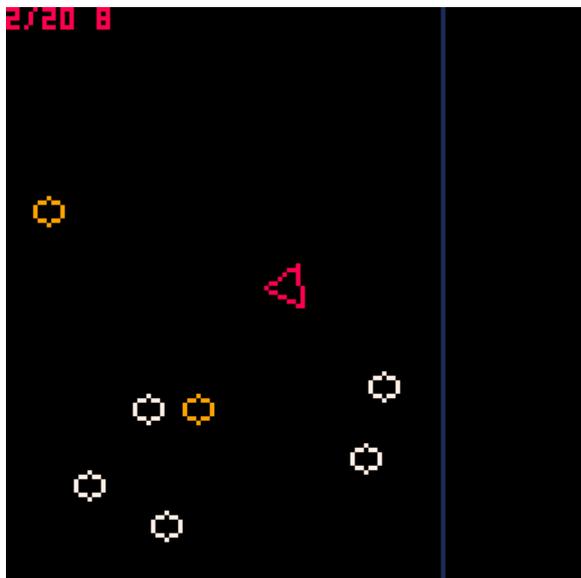


## PICO-8 / LIB\_MATH / L2

### Programmation d'un jeu vidéo

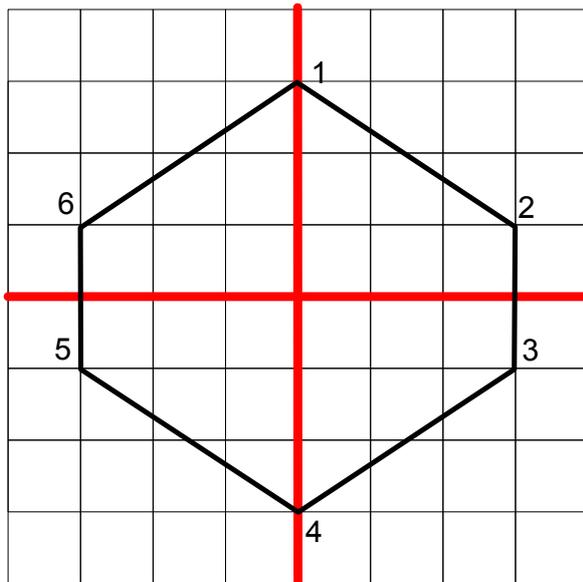
Le but du jeu est de déplacer un véhicule à l'écran pour passer sur un ensemble de vingt rochers éparpillés le plus rapidement possible. L'écran affiche :

- le véhicule (en rouge),
- les rochers en blanc (ou orange s'ils ont été touchés),
- la limite de jeux (en bleu),
- le nombre de rochers sur lesquels le véhicule est déjà passé (ex. 2),
- le nombre total de rochers (ex. 20) et
- l'état du chrono lors du dernier contact (ex. 8).



#### Programme 17 : Un rocher

La première étape consiste à créer la figure permettant de représenter un rocher (à partir des coordonnées cartésiennes des points qui le composent). Le programme ci-après l'affiche pour s'assurer de la conformité de la figure.



Liste des points :

- 1 : {0,3}
- 2 : {3,1}
- 3 : {3,-1}
- 4 : {0,-3}
- 5 : {-3,-1}
- 6 : {-3,1}

```
#include lib_math.p8
rocher=m:figure({
  points={ {0,3}, {3,1}, {3,-1}, {0,-3}, {-3,-1}, {-3,1} },
  segments={ {1,2}, {2,3}, {3,4}, {4,5}, {5,6}, {6,1} }
})

cls()
rocher:trace(7)
```

Programme 18 : Un rocher 20 fois = vingt rochers

Une liste de rochers (nommée "rochers" avec un "s") référence par leur indice (de [1] à [20]) tous les clones de la figure rocher créés puis déplacés de façon aléatoire sur la surface de jeu. Le rocher cloné est centré sur l'origine (en {0,0}). Le déplacement du clone revient donc à le place au coordonnées choisies : { rnd(256) - 128, rnd(256) - 128 }. La fonction rnd(256) retourne une valeur aléatoire (tirée au hasard) entre 0 et 255 (= valeur passée en paramètre de la fonction rnd - 1). Le retrait de 128 à la valeur renvoyée permet d'obtenir une valeur entre -128 (=0-128) et 127 (=255-128). Cela permet de disséminer les rochers sur une surface équivalente à 4 fois celle de l'écran. Ils ne seront donc vraisemblablement pas tous visibles à l'écran.

```
#include lib_math.p8
rocher=m:figure({
  points={ {0,3}, {3,1}, {3,-1}, {0,-3}, {-3,-1}, {-3,1} },
  segments={ {1,2}, {2,3}, {3,4}, {4,5}, {5,6}, {6,1} }
})

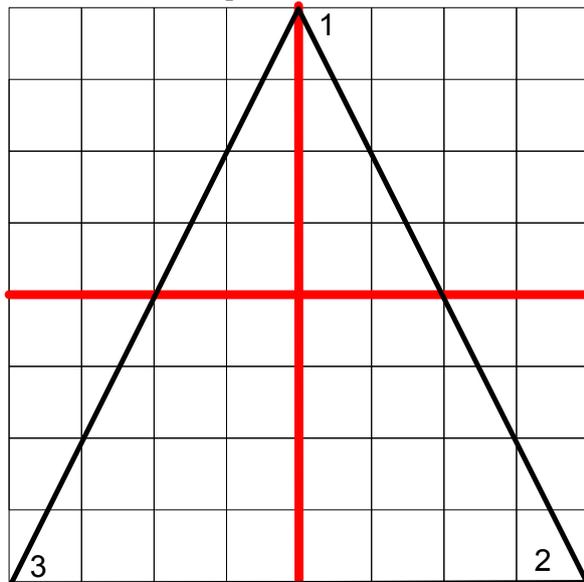
rochers={}
for i=1,20 do
  add(rochers,m:clone(rocher))
  rochers[i]:deplace({
    rnd(256)-128,
    rnd(256)-128
  })
}
```

```
end
```

```
cls()  
for i=1,20 do  
  rochers[i]:trace(7)  
end
```

### Programme 19 : Vingt rochers et un véhicule

L'étape suivante consiste à créer la figure représentant le véhicule. La démarche est similaire à celle de la figure rocher. L'affichage permet de s'assurer de l'absence d'erreur dans les coordonnées des points ou la constitution des segments.



Liste des points :

```
1 : {-4,-4}  
2 : {0,4}  
3 : {4,-4}
```

```
#include lib_math.p8  
rocher=m:figure({  
  points={ {0,3}, {3,1}, {3,-1}, {0,-3}, {-3,-1}, {-3,1} },  
  segments={ {1,2}, {2,3}, {3,4}, {4,5}, {5,6}, {6,1} },  
})
```

```
rochers={}  
for i=1,20 do  
  add(rochers,m:clone(rocher))  
  rochers[i]:deplace({  
    rnd(256)-128,  
    rnd(256)-128  
  })  
end
```

```
vehicule=m:figure({  
  points={ {-4,-4}, {0,4}, {4,-4} },  
  segments={ {1,2}, {2,3}, {3,1} },  
})
```

```
cls()  
for i=1,20 do  
  rochers[i]:trace(7)
```

```
end
vehicule:trace(8)
```

Programme 20 : Vingt rochers et un véhicule qui se déplace...

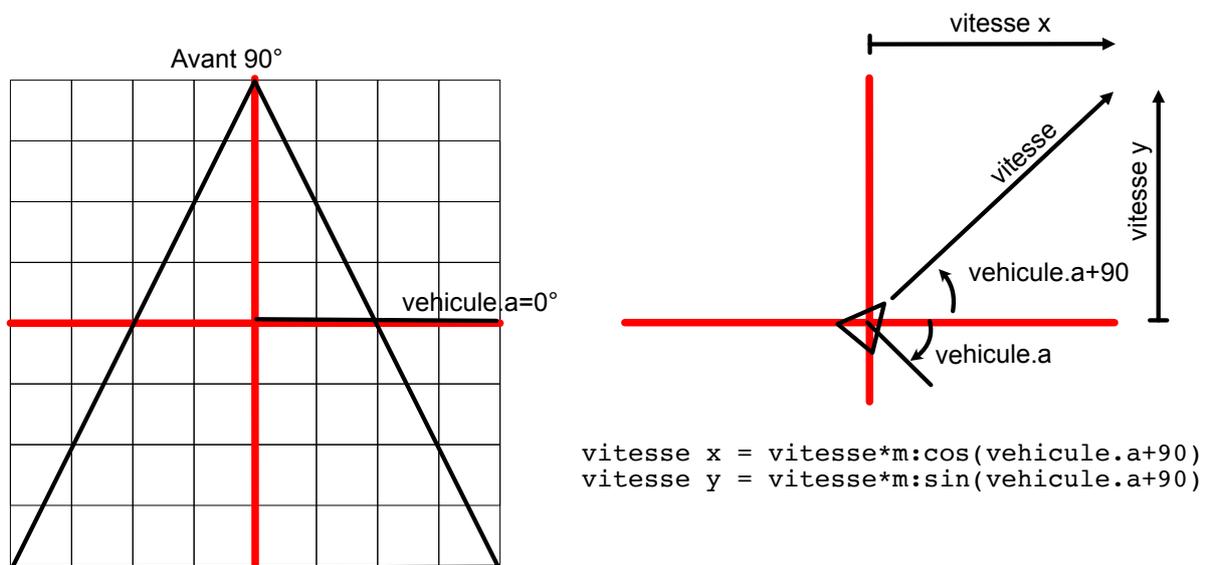
L'ensemble des figures et des clones sont en place. Il est temps de s'intéresser au déplacement du véhicule. Pour cela, il convient de le doter d'une vitesse (fixée à 2). Cette indication complète la description de la figure (vehicule).

Une nouvelle fonction, nommée `_update`, est utilisée. Elle est dédiée à la mise à jour de tous les éléments de contexte à l'exception de ce qui concerne l'affichage (confié à la fonction `_draw`). Le traitement associé à la fonction `_update` teste l'état des touches (haut, gauche, droit) et applique le traitement associé à chaque touche pressée.

Les touches gauche et droite sont utilisées pour faire tourner le véhicule respectivement vers la gauche (sens anti-horaire = `vehicule:tourne(4)`) et vers la droite (sens horaire = `vehicule:tourne(-4)`). L'angle de rotation appliqué est de 4°. La fonction `_update` (comme `_draw`) est appelée 30 fois par seconde. La vitesse de rotation maximale est donc de 120° par seconde (30 x 4).

La touche haut permet de faire avancer le véhicule à la vitesse choisie (2 pixels par appel, soit 60 pixels par seconde). La traversée de l'écran (128 pixels) de gauche à droite (ou de bas en haut) requiert 2,1s et celle de la surface de jeu (256 pixels) approximativement 4,3s. Le déplacement s'effectue selon l'orientation du véhicule.

Pour déplacer le véhicule, il faut déterminer les composantes de sa vitesse sur chaque axe (abscisse, ordonnée). L'angle du véhicule (corrigé pour "pointer" vers l'avant) permet de les calculer à l'aide des fonctions cosinus et sinus [vive la trigonométrie !].



**IMPORTANT :** Pour le calcul du cosinus et du sinus, il faut utiliser les méthodes de la librairie `LIB_MATH` soit `m:cos(angle)` et `m:sin(angle)`. Les fonctions « natives » de PICO-8, nommée `cos` et `sin`, n'utilisent pas des angles en degrés.

```
#include lib_math.p8
```

```

rocher=m:figure({
  points={ {0,3}, {3,1}, {3,-1}, {0,-3}, {-3,-1}, {-3,1} },
  segments={ {1,2}, {2,3}, {3,4}, {4,5}, {5,6}, {6,1} },
})

rochers={}
for i=1,20 do
  add(rochers,m:clone(rocher))
  rochers[i]:deplace({
    rnd(256)-128,
    rnd(256)-128
  })
end

vehicule=m:figure({
  points={ {-4,-4}, {0,4}, {4,-4} },
  segments={ {1,2}, {2,3}, {3,1} },
  vitesse=2
})

function _draw()
  cls()
  for i=1,20 do
    rochers[i]:trace(7)
  end
  vehicule:trace(8)
end

function _update()
  if btn() then
    vehicule:deplace({
      vehicule.vitesse*m*cos(vehicule.a+90),
      vehicule.vitesse*m*sin(vehicule.a+90)
    })
  end
  if btn() then
    vehicule:tourne(4)
  end
  if btn() then
    vehicule:tourne(-4)
  end
end

```

L'ensemble fonctionne correctement. Toutefois, lorsque le véhicule sort de l'écran... et bien... on ne le voit plus. Revenir sur l'écran peut être périlleux, voire impossible.

Programme 21 : ... et une caméra qui le suit.

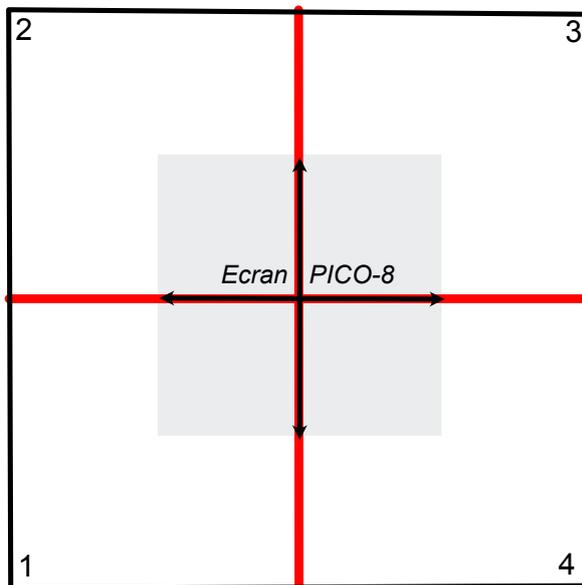
La solution consiste à déplacer la caméra de sorte à maintenir le véhicule au centre de l'écran. L'action se limite à appeler la méthode m:camera avec les coordonnées du point à placer au centre de l'écran. En l'occurrence, il s'agit du point central (p\_central) du véhicule.

ATTENTION : Le `p_central` n'est pas forcément le centre de la figure. Il n'est pas calculé. En réalité, il s'agit du point d'origine `{0, 0}` auquel toutes les transformations appliquées à la figure ont également été appliquées. Lorsque la figure initiale est créée de façon centrée sur l'origine (ce qui est l'usage) alors ce point correspond effectivement au centre de la figure. L'emploi de la méthode `:centre` permet de le modifier (pour choisir le centre d'une rotation à appliquer par exemple).

```
...
function _draw()
  cls()
  m:camera(vehicule.p_central)
  for i=1,20 do
    rochers[i]:trace(7)
  end
  vehicule:trace(8)
end
end
...
```

#### Programme 22 : Délimiter la surface de jeu

La caméra suivant le véhicule, les limites de l'écran ne délimitent plus l'espace de jeu. Par conséquent, il est possible d'en sortir et de se perdre. L'ajout d'une frontière visible délimitant l'aire de jeu permet au joueur de limiter ses déplacements à celle-ci.



Liste des points :

- 1 : `{-128,-128}`
- 2 : `{-128,128}`
- 3 : `{128,128}`
- 4 : `{128,-128}`

```
...
vehicule=m:figure({
  points={ {-4,-4}, {0,4}, {4,-4} },
  segments={ {1,2}, {2,3}, {3,1} },
  vitesse=2
})
```

```
limite=m:figure({
```

```

points={ {-128,-128}, {-128,128}, {128,128}, {128,-128} },
segments={ {1,2}, {2,3}, {3,4}, {4,1} }
})

```

```

function _draw()
  cls()
  m:camera(vehicule.p_central)
  limite:trace(1)
  for i=1,20 do
    rochers[i]:trace(7)
  end
  vehicule:trace(8)
end
...

```

Programme 23 : Manhattan !

Pour détecter si le véhicule passe sur un rocher déterminé, il faut évaluer sa distance par rapport à celui-ci. La distance n'est pas évaluée de bord à bord mais entre les points centraux (`p_central`) de chaque figure. La distance correspond donc à l'hypoténuse d'un triangle rectangle dont la longueur de chacun des deux autres côtés est respectivement la valeur absolue de la différence des abscisses (des points centraux) et la différence des ordonnées (des points centraux). L'application du théorème de Pythagore permet de calculer la distance.

**Théorème de Pythagore** : Si un triangle ABC est rectangle en C, alors  $AB^2 = AC^2 + BC^2$ .

Avec  $AB = c$ ,  $AC = b$  et  $BC = a$  (cf. figure ci-dessous), la formule s'écrit :  $a^2 + b^2 = c^2$ . La distance  $c$  se calcule donc de la façon suivante :  $c = \sqrt{a^2 + b^2}$

Les coordonnées d'un (m:)point sont stockées dans une liste de la forme { *abscisse, ordonnée* }. L'expression `point[1]` renvoie son abscisse (premier élément de la liste). L'expression `point[2]` renvoie son ordonnée (second élément de la liste).

A titre d'exemple, le programme ci-après affiche les valeurs 5 et 7.

```

#include lib_math.p8
x=5
y=7
point=m:point(x,y)
print(point[1])
print(point[2])

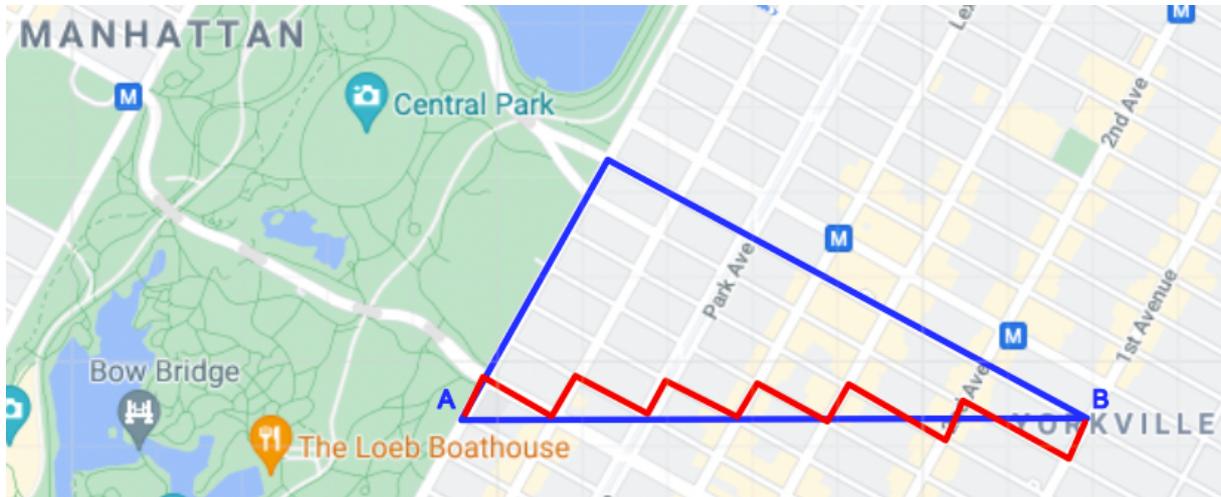
```

Le point A est le point central de la figure "vehicule", ses coordonnées d'abscisse et d'ordonnées sont : `vehicule.p_central[1]` et `vehicule.p_central[2]`.

Le point B est le point central de la figure "rocher" pour laquelle la distance doit être calculée. Pour abrégé, il est nommé "r". Ses coordonnées d'abscisse et d'ordonnée sont : `r.p_central[1]` et `r.p_central[2]`.



a). Lorsqu'un résultat approximatif est suffisant, l'approche mathématique (exacte) est abandonnée au profit d'une approximation appelée "distance de Manhattan" (car mise au point par les chauffeurs de taxi de Manhattan). Manhattan est un quartier de la ville de New-York (aux États-Unis). Ses rues sont agencées selon un quadrillage.



©Google maps

Pour se rendre du point A au point B, les taxis ne peuvent pas traverser les bâtiments. Ils doivent rouler sur la route et longer le côté des bâtiments. La distance réellement parcourue par le taxi correspond alors à la somme de la longueur des deux côtés adjacents à l'angle droit (et non à l'hypoténuse).

La distance de Manhattan est donc la somme de la valeur absolue des différences d'abscisse et d'ordonnée. Soit sous une forme moins littéraire et plus mathématique :

$$\begin{aligned} \text{distance manhattan} \\ &= |r.p\_central[1] - vehicule.p\_central[1]| \\ &+ |r.p\_central[2] - vehicule.p\_central[2]| \end{aligned}$$

La fonction `distance_manhattan` effectue ce calcul pour le rocher "r" passé en paramètre et renvoie (return) le résultat.

```
#include lib_math.p8
rocher=m:figure({
  points={ {0,3}, {3,1}, {3,-1}, {0,-3}, {-3,-1}, {-3,1} },
  segments={ {1,2}, {2,3}, {3,4}, {4,5}, {5,6}, {6,1} },
  touche=0
})
```

...

```
limite=m:figure({
  points={ {-128,-128}, {-128,128}, {128,128}, {128,-128} },
  segments={ {1,2}, {2,3}, {3,4}, {4,1} }
})
```

```
function distance_manhattan(r)
```

```

return
  abs(r.p_central[1] - vehicule.p_central[1]) +
  abs(r.p_central[2] - vehicule.p_central[2])
end

```

La fonction *trace* analyse la situation du véhicule pour le rocher "r" passé en paramètre. Elle utilise la fonction *distance\_manhattan* pour calculer sa distance (approximative). Si elle est inférieure à 5, le véhicule touche le rocher "r" (et si ce dernier n'a pas déjà été touché ⇔ `r.touche==0`) alors le rocher est considéré comme "touché" : `r.touche=1`.

Les rochers touchés sont affichés en orange (`r:trace(9)`) les autres en blanc (`r:trace(7)`).

```

function trace(r)
  if distance_manhattan(r)<5 then
    if r.touche==0 then
      r.touche=1
    end
  end
  if r.touche==1 then r:trace(9) else r:trace(7) end
end

```

```

function _draw()
  cls()
  m:camera(vehicule.p_central)
  limite:trace(1)
  foreach(rochers, trace)
  vehicule:trace(8)
end
...

```

L'instruction **foreach** parcourt tous les éléments d'une liste (*rochers*) et appelle la fonction associée (*trace*).

```

foreach(rochers, trace)
est équivalent à :
for i=1,#rochers do
  trace(rochers[i])
end

```

Exercice : remplacer la fonction *distance\_manhattan* par *distance\_pythagore* ci-dessous (penser à modifier la fonction *trace* !):

```

function distance_pythagore(r)
  return sqrt(
    (r.p_central[1] - vehicule.p_central[1])^2 +
    (r.p_central[2] - vehicule.p_central[2])^2
  )
end

```

Programme 24 : Niveau, compteur et minuteur

La dernière étape consiste à afficher un compteur de rochers touchés et la valeur du minuteur lors du dernier contact avec un rocher (non encore touché). L'instruction **srand** est utilisée pour fixer la "graine" (valeur de départ pour le calcul) du générateur de nombre pseudo-aléatoire. Cela permet de recommencer un "niveau" avec la même disposition des rochers (sinon elle changerait à chaque lancement du programme). Pour changer de niveau, il suffit de modifier la valeur du **srand** : `srand(0)`, `srand(1)`, `srand(2)`, ...

```
#include lib_math.p8
srand(0)
compteur,minuteur=0,0

rocher=m:figure({
  points={ {0,3}, {3,1}, {3,-1}, {0,-3}, {-3,-1}, {-3,1} },
  segments={ {1,2}, {2,3}, {3,4}, {4,5}, {5,6}, {6,1} },
  touche=0
})

...

function trace(r)
  if distance_manhattan(r)<5 then
    if r.touche==0 then
      compteur+=1
      minuteur=flr(t())
      r.touche=1
    end
  end
  if r.touche==1 then r:trace(9) else r:trace(7) end
end

function _draw()
  cls()
  m:camera({0,0}) print(compteur.."/20 " ..minuteur)
  m:camera(vehicule.p_central)
  limite:trace(1)
  foreach(rochers,trace)
  vehicule:trace(8)
end

...
```

La caméra est placée en  $\{ 0, 0 \}$  pour afficher les indications (compteur, minuteur) à l'écran toujours au même endroit (en haut à gauche) sinon elles se déplacent avec l'ensemble du plan (contenant les rochers).

Exercice : Retirer **m:camera({0,0})** de la fonction `_draw` et observer le résultat.

Le programme complet (programme24.p8) :

```
#include lib_math.p8
srand(0)
compteur,minuteur=0,0

rocher=m:figure({
  points={ {0,3}, {3,1}, {3,-1}, {0,-3}, {-3,-1}, {-3,1} },
  segments={ {1,2}, {2,3}, {3,4}, {4,5}, {5,6}, {6,1} },
  touche=0
})

rochers={}
for i=1,20 do
  add(rochers,m:clone(rocher))
  rochers[i]:deplace({
    rnd(256)-128,
    rnd(256)-128
  })
end

vehicule=m:figure({
  points={ {-4,-4}, {0,4}, {4,-4} },
  segments={ {1,2}, {2,3}, {3,1} },
  vitesse=2
})

limite=m:figure({
  points={ {-128,-128}, {-128,128}, {128,128}, {128,-128} },
  segments={ {1,2}, {2,3}, {3,4}, {4,1} }
})

function distance_manhattan(r)
  return
  abs(r.p_central[1] - vehicule.p_central[1]) +
  abs(r.p_central[2] - vehicule.p_central[2])
end

function trace(r)
  if distance_manhattan(r)<5 then
    if r.touche==0 then
      compteur+=1
      minuteur=flr(t())
      r.touche=1
    end
  end
end
```

```
if r.touche==1 then r:trace(9) else r:trace(7) end  
end
```

```
function _draw()  
  cls()  
  m:camera({0,0}) print(compteur.."/20 " ..minuteur)  
  m:camera(vehicule.p_central)  
  limite:trace(1)  
  foreach(rochers,trace)  
  vehicule:trace(8)  
end
```

```
function _update()  
  if btn() then  
    vehicule:deplace({  
      vehicule.vitesse*m:cos(vehicule.a+90),  
      vehicule.vitesse*m:sin(vehicule.a+90)  
    })  
  end  
  if btn() then  
    vehicule:tourne(4)  
  end  
  if btn() then  
    vehicule:tourne(-4)  
  end  
end
```

## PICO-8

### Langage de programmation (LUA)

#### Principales instructions

##### **rnd**(*donnée*)

- Renvoie un nombre pseudo-aléatoire  $n$ , qui vérifie  $0 \leq n < \text{donnée}$   
Si vous souhaitez un nombre entier, utilisez **flr (rnd (donnée) )** .

##### **srnd**(*donnée*)

- Fixe la graine du générateur de nombre pseudo-aléatoire.  
La graine est automatiquement définie au démarrage de la cartouche.

##### *variable1, variable2 = valeur1, valeur2*

- AFFECTE la valeur1 à la variable1 et la valeur2 à la variable2  
(à ne pas confondre avec l'égalité booléenne ==).

##### **foreach**(*liste, fonction*)

- Pour chaque élément de la liste, appelle la fonction avec l'élément comme seul paramètre.

##### **function \_update()** *traitement* **end**

- Exécute le *traitement* 30 fois par seconde (préalablement à l'appel de la fonction `_draw`).

#### Exemples

##### Afficher les nombres d'une liste

```
cls()  
srnd(10269)  
liste, nombre={}, 10  
for i=1, nombre do  
  add(liste, flr(rnd(100)))  
end  
foreach(liste, print)
```