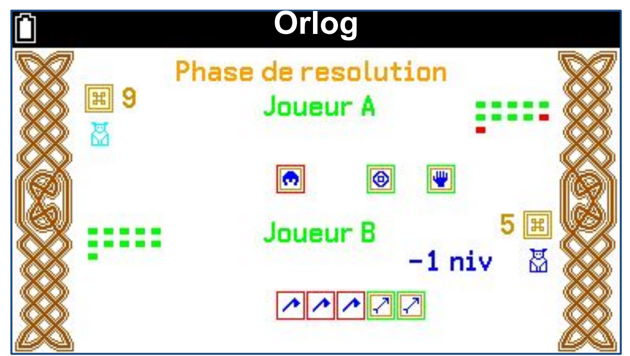
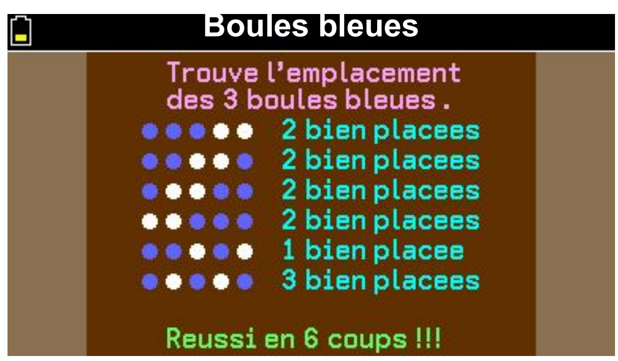
Atelier jeu vidéo
Quelques-uns de nos jeux
Python pour Graph 90+E et
Graph 35+E II :
Saute-Moutons (rentrée 2020)
SynchroD (concours rentrée 2021)
RPG Alrys (concours rentrée 2021)
Doom-like Pykaster3D (concours 2022)
FlappyBird (atelier 1er trimestre 2022)
1000 Bornes (concours 2023-2024)
Etiord City (atelier 1er trimestre 2023)
Gravity Guy (atelier 1er trimestre 2023)
[AC]: Battre des ailes
249%+1
94pts
Gauche
200 km
200 pts
0 km
100 pts
VP
87

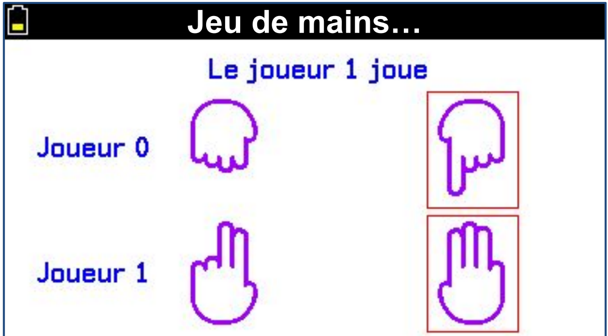
Atelier jeu vidéo

Création d'un jeu vidéo **Python** pour calculatrice **Casio Graph Math+**

Exemples de jeux **Python** pour **Graph Math+** par **Florian Allard** :







… Cubefield !

Jeu en langage Flash sorti en 2006 pour navigateurs Internet par Max Abernethy

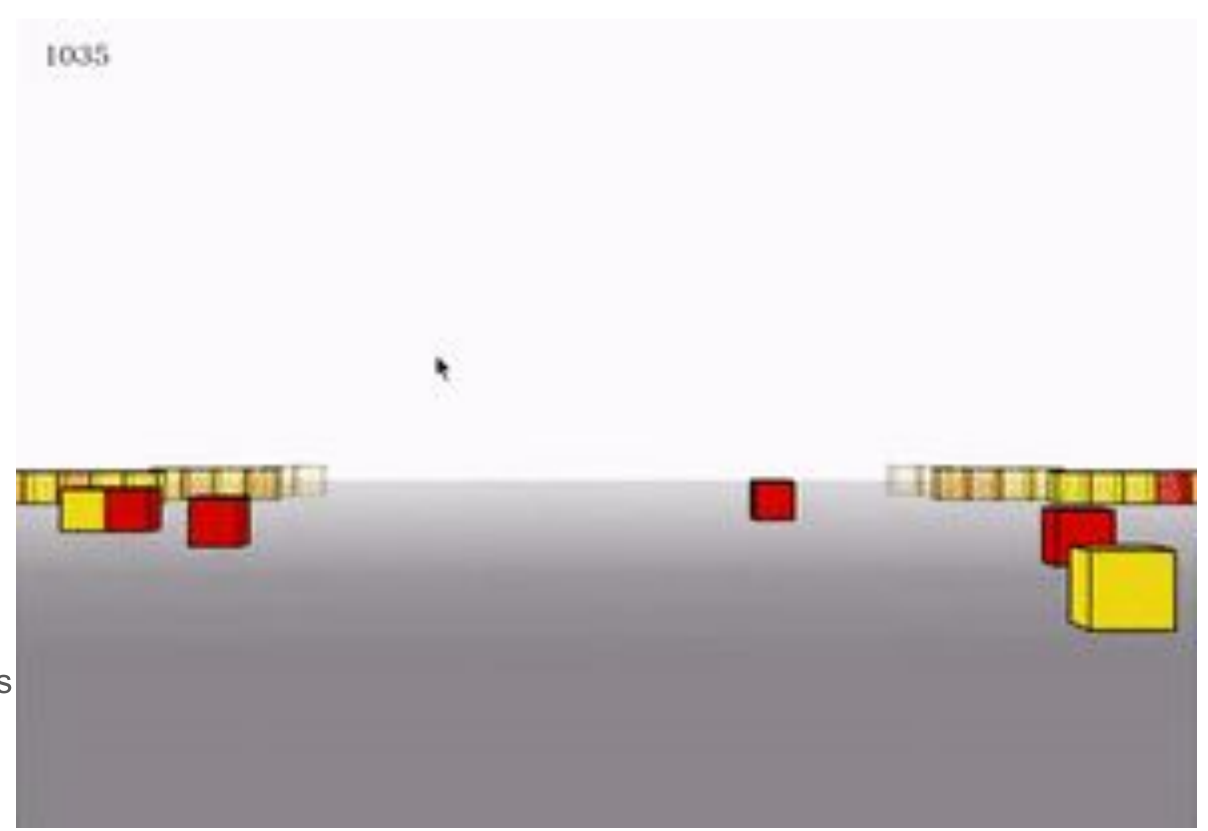
Principe : piloter un petit vaisseau à travers un champ de cubes

Objectifs :

- éviter les cubes
- aller aussi loin que possible

Affichage : 3D *“à la 3e personne”*

Graphismes : minimalistes avec des formes géométriques simples aux couleurs unies



Contrôles : 2 actions possibles *(gauche + droite)*

Composition d’un jeu vidéo

3 étapes qui tournent en boucle :

- **gestion des entrées** :
  réagir aux pressions de touches

- **moteur physique** :
  simulation de l’environnement et de la physique du jeu

- **moteur graphique** :
  dessin à l’écran de la scène du jeu

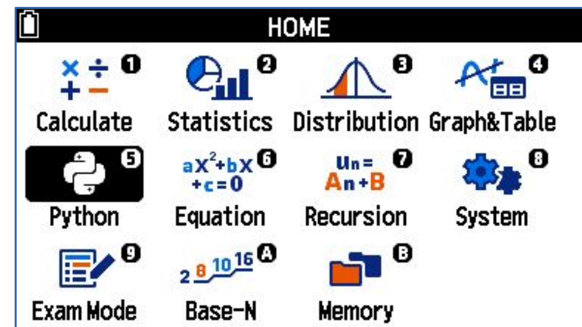
… répéter 10-30 fois par seconde !

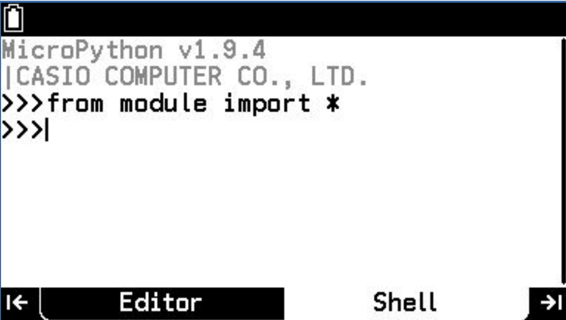
L'application Python Graph Math+

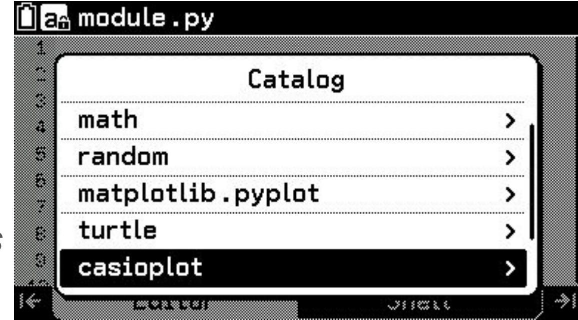
Interpréteur **MicroPython** en version **1.9.4**

5 bibliothèques intégrées :

- **3** bibliothèques standard **Python 3** :
    - **math**
      fonctions mathématiques
    - **random**
      génération de données aléatoires
    - **turtle**
      tracé par déplacements (à la Scratch / Logo)
- **1** bibliothèque tierce populaire :
    - **matplotlib** (limitée à **matplotlib.pyplot**)
      tracé de diagrammes dans des repères
- **1** bibliothèque spécifique à **Casio**
    - **casioplot**
      tracé par pixels (utilisé par turtle et matplotlib) + test de touches



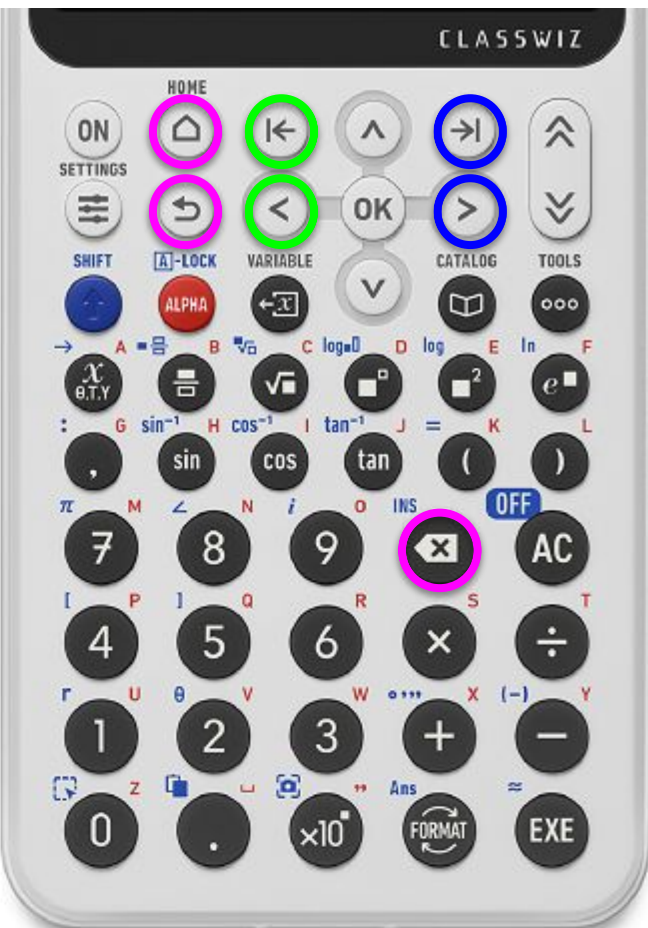




Choix touches de contrôle Cubefield

Au moins 3 touches :

- pour décaler le vaisseau sur la gauche :
  une touche indiquant une direction vers la gauche
- pour décaler le vaisseau sur la droite :
  une touche indiquant une direction vers la droite
- pour quitter le jeu :
  une touche indiquant une une sortie ou un retour en arrière



La fonction casioplot.getkey()

Test non bloquant.

Renvoie **immédiatement** une valeur :

- ➢ si aucune touche pressée, **None**
- ➢ sinon, **code de la touche**

Avantage : script Python non figé lors des tests de touches
(essentiel pour les jeux ne se jouant pas au tour par tour)

Pour déterminer les codes de touches, créons une fonction **waitkey()** :

```
waitkey.py
1  from casioplot import getkey
2
3  def waitkey():
4    key = None
5    while key == None:
6      key = getkey()
7    return key
8
9
```

Editor | Shell

Catégorie	Touche	Code
directions	<	23
	>	25
	∧	14
	∨	34
	⇤	13
	⇥	15
	⇞	16
	⇟	26
validation actions	EXE	95
	OK	24
	⇧	31
	ALPHA	32
retour annulation quitter	⌂	12
	↶	22
	⌫	64

getkey() et codes de touches

Le code retourné par **getkey()** obéit à 3 règles :

- nombre entier à 2 chiffres
- en dizaine, numéro de rangée de la touche depuis le haut
- en unité, numéro de touche sur la rangée depuis la gauche

2 touches ne sont pas testables via **getkey()** :

- **[ON]** qui ne renvoie pas 11 mais est ignorée
- **[AC]** qui ne renvoie pas 65 mais interrompt le script

Retenons et définissons :

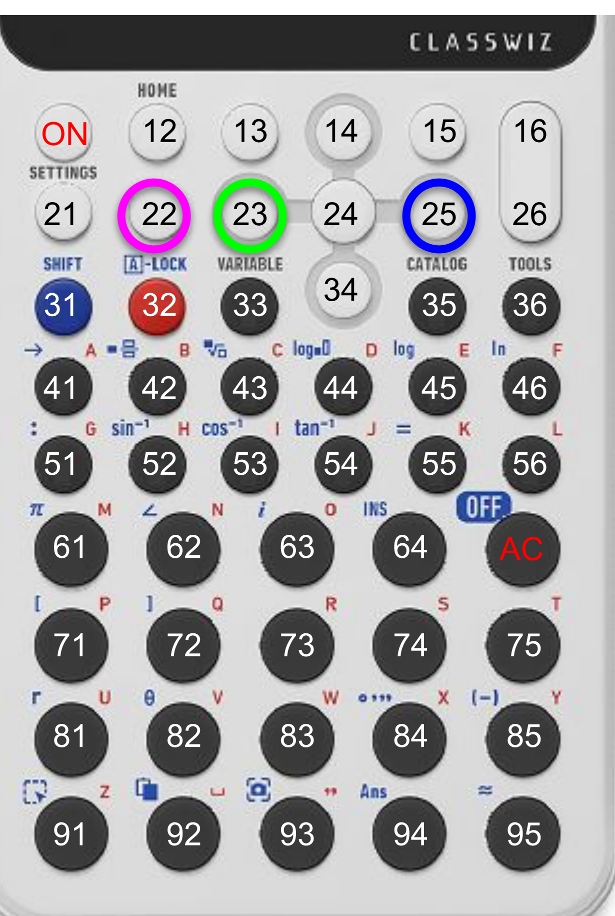
- K_ESC = 22
- K_LEFT = 23
- K_RIGHT = 25

Boucle principale du jeu :

```
from casioplot import *

K_ESC, K_LEFT, K_RIGHT = 22, 23, 25

key = None
while key != K_ESC:
  #...
  key = getkey()
```



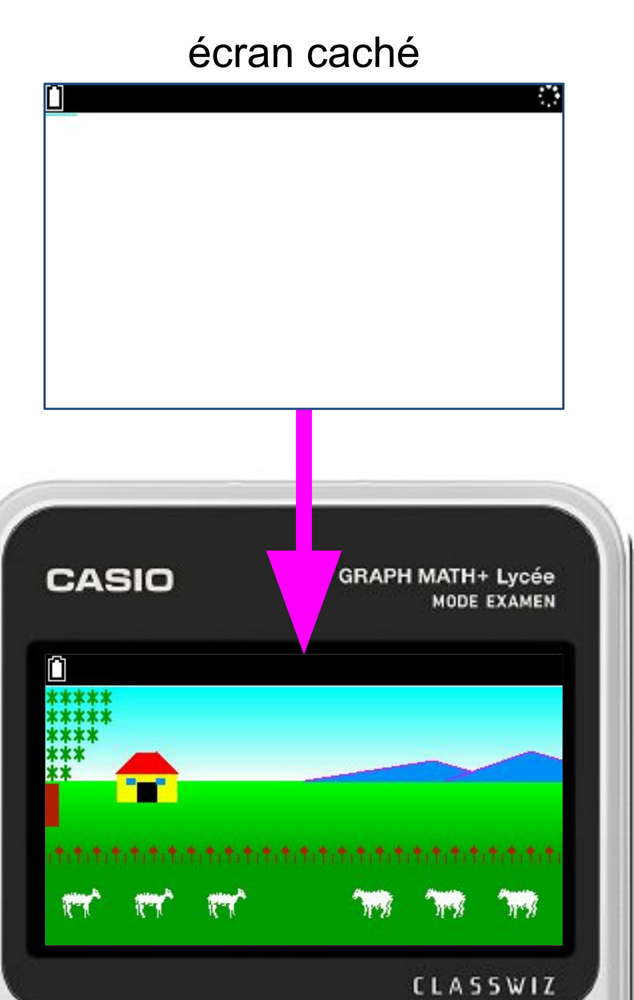
Casioplot et principe d'affichage

Fonctionnement en **double buffering** :

- les divers tracés demandés ne sont pas affichés, mais écrits dans une zone mémoire dite ***"écran caché"***
- sur demande, copie de l' ***"écran caché"*** vers la ***"mémoire écran"***, affichant tous les derniers changements

Avantages :

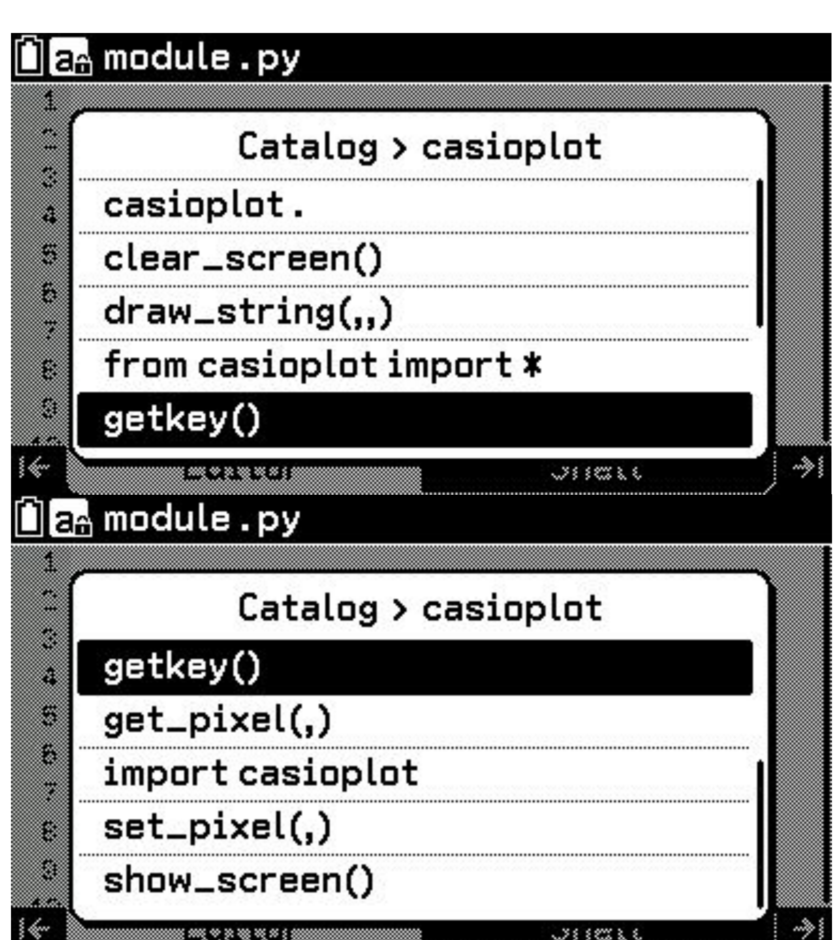
- masque affichages intermédiaires inesthétiques entre deux écrans (tracés partiels, clignotements, etc.)
- fluidité des animations



Casioplot et fonctions graphiques

5 fonctions graphiques intégrées :

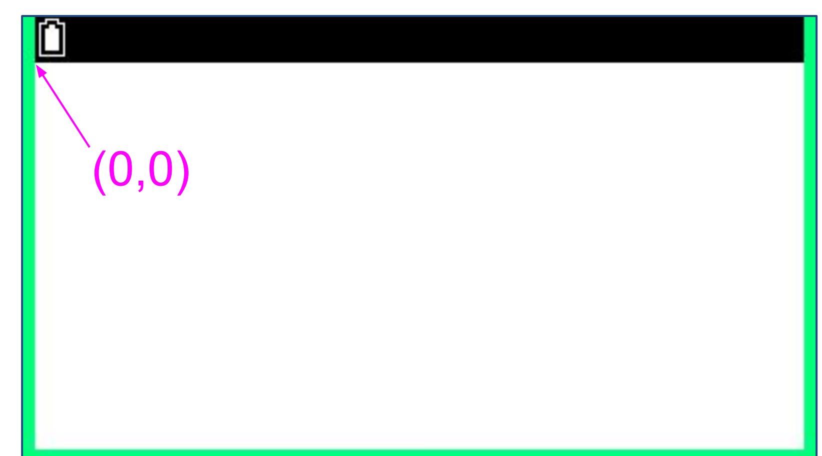
- **4** fonctions agissant sur l' ***écran caché*** :
  - **clear_screen()**
    Efface en blanc
  - **draw_string(x,y,texte,couleur,taille)**
    Écrit des chaînes de caractères
  - **get_pixel(x,y)**
    Retourne la couleur d'un pixel
  - **set_pixel(x,y,couleur)**
    Change la couleur d'un pixel
- **1** fonction agissant sur l'**écran** :
  - **show_screen()**
    Affiche ce qui a été préparé en écran caché



Casioplot et zone graphique

Plusieurs zones écran réservées non modifiables :

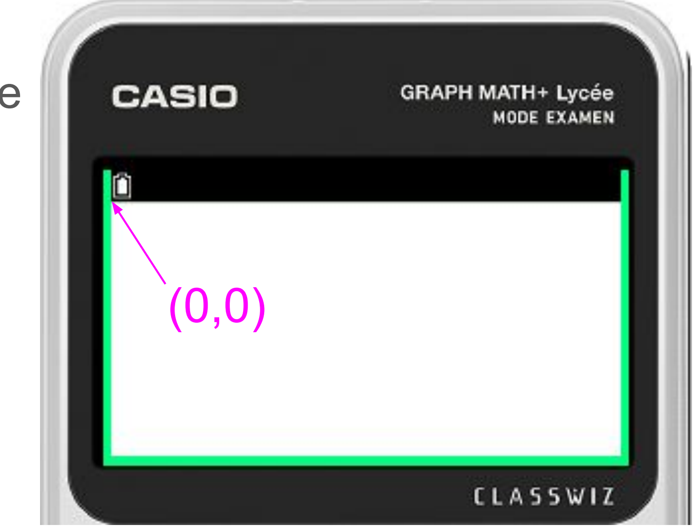
- En haut, barre de statut de 24 pixels de haut
- Marges du mode examen, en noir ou vert :
  - 6 pixels de large à gauche et à droite
  - 8 pixels de haut en bas



Le reste en blanc ci-contre, dit **“zone graphique”** est librement contrôlable.

Coordonnées de pixels à partir du coin supérieur gauche de la zone :

- abscisse **x** de gauche à droite
- ordonnée **y** de haut en bas



Coordonnées 1er pixel en haut à gauche de la zone :

x=0 et **y=0**, noté **(0,0)**

Mesure zone graphique “à la tortue”

Idée : get_pixel() renvoie None si appelé avec des coordonnées hors zone

Principe :

- On se positionne en haut à gauche en (0,0)
- Tant que get_pixel() renvoie autre chose que None :
  - Déplacement vers la droite : x += 1
  - Déplacement vers le bas : y += 1

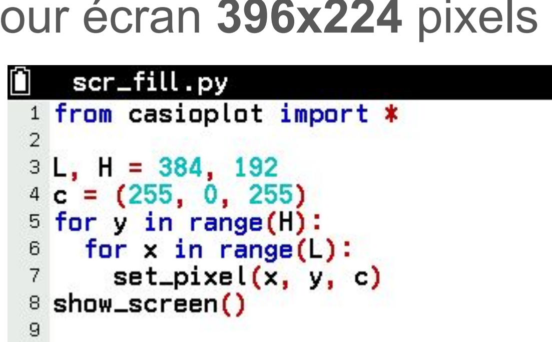
screen.py

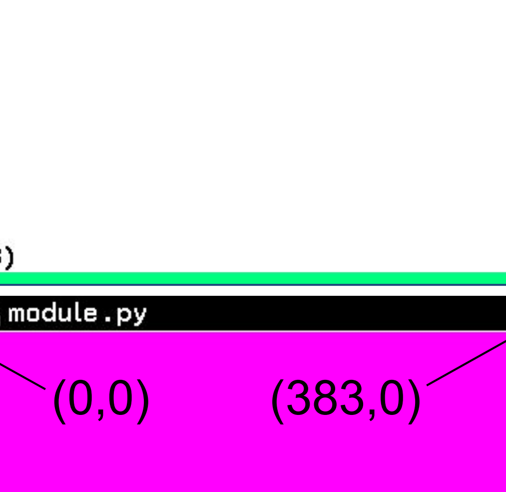
```
def scr_size():
  x, y, dx, dy = 0, 0, 1, 1
  while dx or dy:
    set_pixel(x, y, (255, 0, 0))
    show_screen()
    x, y = x + dx, y + dy
    if get_pixel(x, y) == None:
      if dx and get_pixel(x-1, y):
        x, dx = x - 1, 0
      if dy and get_pixel(x, y-1):
        y, dy = y - 1, 0
  return x + 1, y + 1
```

Editor Shell

Zone graphique **384x192** pixels pour écran **396x224** pixels

Exemples	Coordonnées
coin supérieur gauche	(0, 0)
coin supérieur droit	(383, 0)
coin inférieur gauche	(0, 191)
coin inférieur droit	(383, 191)





Casioplot et format couleur

Listes décomposant en 3 couleurs primaires : **(Rouge, Vert, Bleu)**

Chaque élément est l’intensité de la couleur primaire ; un entier **8 bits** *(2^8= **256** valeurs différentes)* :

- minimum **0** *(absence de la couleur)*
- maximum **255**

La couleur résultante est la synthèse additive des 3 composantes.

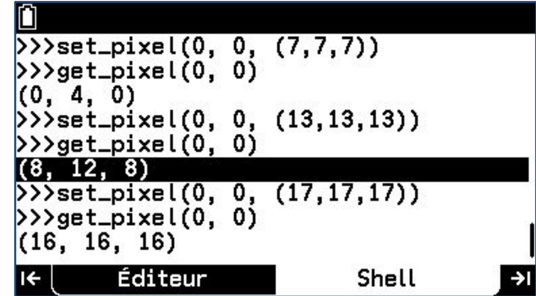
Codage sur 8+8+8= **24 bits**, permettant de décrire 2^{24}= **16 777 216** couleurs différentes.

exemples de couleurs	composante rouge	composante verte	composante bleue	liste pour casioplot
rouge	255	0	0	(255,0,0)
vert	0	255	0	(0,255,0)
bleu	0	0	255	(0,0,255)
jaune	255	255	0	(255,255,0)
magenta	255	0	255	(255,0,255)
cyan	0	255	255	(0,255,255)
blanc	255	255	255	(255,255,255)
gris clair	191	191	191	(191,191,191)
gris	127	127	127	(127,127,127)
gris foncé	63	63	63	(63,63,63)
noir	0	0	0	(0,0,0)
orange	255	127	0	(255,127,0)
violet	127	0	255	(127,0,255)
marron	127	63	0	(127,63,0)

Ecran Graph Math+ et couleurs

L'écran **Graph Math+** affiche les couleurs les plus proches sur l'échelle **RVB 565**, codant :

- la rouge sur 5 bits
  (2^5= 32 teintes différentes, multiples de 8)
- le verte sur 6 bits
  (2^6= 64 teintes différentes, multiples de 4)
- la bleu sur 5 bits
  (2^5= 32 teintes différentes, multiples de 8)



Par combinaison affichage sur 5+6+5= **16 bits** permettant 2^{16}= **65536** couleurs différentes

module.py

```
1 from casioplot import *
2 L, H = 384, 192
3 def dnoir(y0, h, c):
4   for x in range(L):
5     cd = [c[k] * x // (L - 1)
6           for k in range(3)]
7     for dy in range(h):
8       set_pixel(x, y0 + dy, cd)
9   show_screen()
```

Editor | Shell

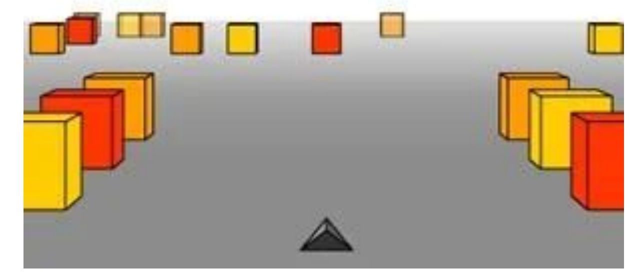
dnoir(0, 64, (255, 0, 0))

dnoir(64, 64, (0, 255, 0))

dnoir(128, 64, (0, 0, 255))

Choix graphiques pour Cubefield

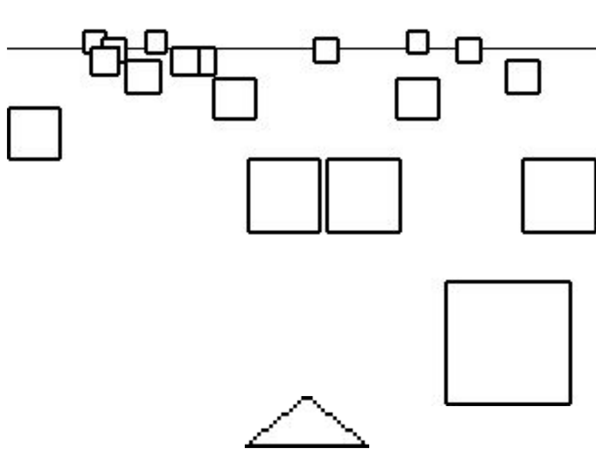
Tout le dessin (sauf texte) passe par **set_pixel()**.
Donc on moins on modifie de pixels, plus ça ira vite.



Simplifications pour la Graph Math+ :

- Cubes : juste la face avant (non remplie)
- Sol : juste une ligne d’horizon
- Le vaisseau : triangle (non rempli)

On dessinera donc : des lignes, carrés, et triangles.



Extension casioplot avec draw_line #1

Étendons **casioplot** avec une fonction pour tracer un segment entre 2 points.

Spécifications : draw_line(x1, y1, x2, y2, c), avec :

- **x1, y1** : coordonnées d'une des extrémités du segment
- **x2, y2** : coordonnées de l'autre extrémité du segment
- **c** : la couleur du trait

Idée 1 : "approche naïve"
Allumer tous les points d'abscisse entière du se²gment.
Pour tout entier $\mathbf{x \in [x_1, x_2]}$:
  $\mathbf{y \leftarrow y_1 + (y_2 - y_1) \times (x - x_1) / (x_2 - x_1)}$
  allumer le pixel **(x, y)**


```
dlin1.py
1 from casioplot import *
2
3 def draw_line(x1, y1, x2, y2, c):
4  if x2 < x1:
5   x1, y1, x2, y2 = x2, y2, x1, y1
6  for x in range(x1, x2 + 1):
7   y=y1+(y2-y1)*(x-x1)/((x2-x1)or 1)
8   set_pixel(round(x), round(y), c)
9
```


Problème : tracé discontinu des segments trop inclinés…

Extension casioplot avec draw_line #2

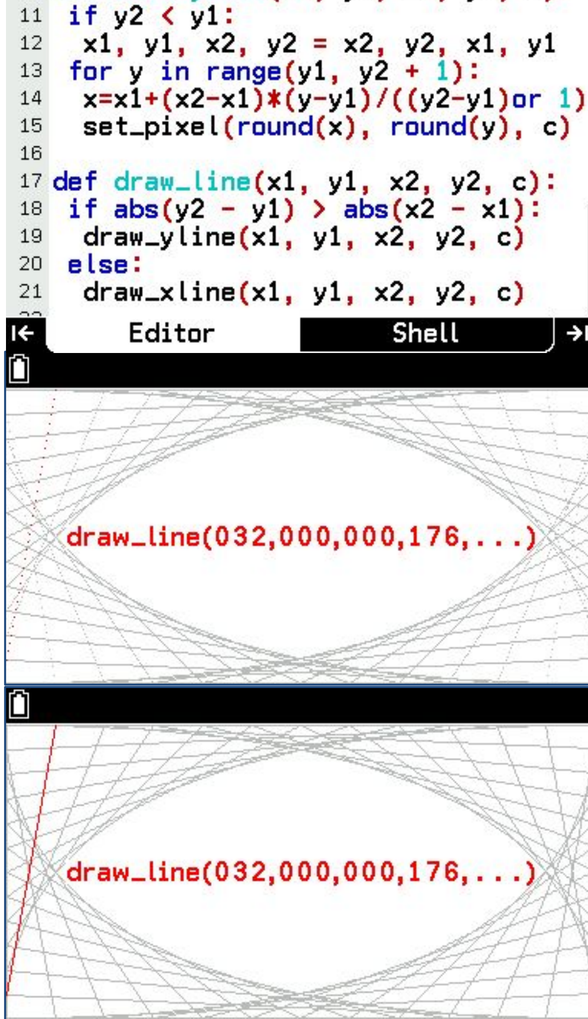
Idée 2 : “2 cas séparés”

- Renommons la fonction précédente qui allume tous les points d’abscisses entières en **draw_xline()**
- Créons une fonction similaire **draw_yline()** qui allume tous les points d’ordonnées entières du segment :

  Pour tout entier $\mathbf{y} \in [\mathbf{y_1},\mathbf{y_2}]$:

  $\mathbf{x} \leftarrow \mathbf{x_1}+(\mathbf{x_2}-\mathbf{x_1})\times(\mathbf{y}-\mathbf{y_1})/(\mathbf{y_2}-\mathbf{y_1}))$

  allumer le pixel (*arrondi*(**x**), **y**)
- Et créons une fonction **draw_line()** qui appelle :
  - si |pente|<1 : **draw_xline()**
  - si |pente|>1 : **draw_yline()**



Problème : Temps de tracé sur Graph Math+ : **42,07s**

À la recherche du temps perdu…

Qu'est-ce qui prends du temps à un processeur ?

- Les calculs répétés dans les boucles
- Certains opérateurs

  par rapidité croissante :
  - *Diviser*
  - *Multiplier*
  - *Additionner/Soustraire*
- Certains types d'opérandes

  par rapidité croissante :
  - *nombres flottants (float)*
  - *nombres entiers (int)*

Optimisation draw_line #1

Idée 1 : “calculs constants hors boucle”

Par exemple dans la boucle **draw_xline()** :

- Nous calculons $\mathbf{y=y_1+(y_2-y_1)\times(x-x_1)/(x_2-x_1)}$
- Alors que $\mathbf{(y_2-y_1)/(x_2-x_1)}$ est constant
- Et nous arrondissons **x**

Pour optimiser :

- ➢ Calculons 1 fois hors boucle $\mathbf{m=(y_2-y_1)/(x_2-x_1)}$
- ➢ Remplaçons par $\mathbf{y=y_1+(x-x_1)\times m}$
- ➢ Arrondissons $\mathbf{x_1}$ et $\mathbf{x_2}$ hors boucle pour que **x** prenne forcément des valeurs entières
- ➢ Et faisons pareil pour **draw_yline()**

dlin2.py

```
1 from casioplot import *
2
3 def draw_xline(x1, y1, x2, y2, c):
4  if x2 < x1:
5   x1, y1, x2, y2 = x2, y2, x1, y1
6  for x in range(x1, x2 + 1):
7   y=y1+(y2-y1)*(x-x1)/((x2-x1)or 1)
8   set_pixel(round(x), round(y), c)
9
```

```
17 def draw_line(x1, y1, x2, y2, c):
18  if abs(y2 - y1) > abs(x2 - x1):
19   draw_yline(x1, y1, x2, y2, c)
20  else:
21   draw_xline(x1, y1, x2, y2, c)
22
23
24
25
```

Pour tout entier $\mathbf{x\in[x_1,x_2]}$:

$\mathbf{y\leftarrow y_1+(y_2-y_1)\times(x-x_1)/(x_2-x_1)}$
allumer pixel (**x**, *arrondi*(**y**))

dlin3.py

```
1 from casioplot import *
2
3 def draw_xline(x1, y1, x2, y2, c):
4  if x2 < x1:
5   x1, y1, x2, y2 = x2, y2, x1, y1
6  m = (y2 - y1) / ((x2 - x1) or 1)
7  for x in range(x1, x2 + 1):
8   y = y1 + (x - x1)*m
9   set_pixel(x, round(y), c)
```

```
19 def draw_line(x1, y1, x2, y2, c):
20  x1, y1, x2, y2 = [
21   round(v)
22   for v in (x1, y1, x2, y2)
23  ]
24  if abs(y2 - y1) > abs(x2 - x1):
25   draw_yline(x1, y1, x2, y2, c)
26  else:
27   draw_xline(x1, y1, x2, y2, c)
```

$\mathbf{m\leftarrow(y_2-y_1)/(x_2-x_1)}$
Pour tout entier $\mathbf{x\in[x_1,x_2]}$:

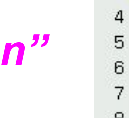
$\mathbf{y\leftarrow y_1+(x-x_1)\times m}$
allumer pixel (**x**, *arrondi*(**y**))

Temps de tracé sur calculatrice Graph Math+ :
42,07s → **35,57s** *(-15%)*

draw_line(032,000,000,176,...)

Optimisation draw_line #2

Idée 2 : “supprimer la multiplication”



Par exemple dans la boucle **draw_xline()**

- Nous calculons $\mathbf{y=y_1+(x-x_1)\times m}$
- À chaque itération, **x** augmente de **1**
- Donc **y** augmente de **m**

dlin3.py

```
1 from casioplot import *
2
3 def draw_xline(x1, y1, x2, y2, c):
4  if x2 < x1:
5   x1, y1, x2, y2 = x2, y2, x1, y1
6  m = (y2 - y1) / ((x2 - x1) or 1)
7  for x in range(x1, x2 + 1):
8   y = y1 + (x - x1)*m
9   set_pixel(x, round(y), c)
```

Editor | Shell

$m \leftarrow (y_2-y_1)/(x_2-x_1)$
Pour tout entier $x \in [x_1,x_2]$:
  $y \leftarrow y_1+(x-x_1)\times m$
  allumer pixel (**x**,*arrondi*(**y**))

dlin4.py

```
1 from casioplot import *
2
3 def draw_xline(x1, y1, x2, y2, c):
4  if x2 < x1:
5   x1, y1, x2, y2 = x2, y2, x1, y1
6  m = (y2 - y1) / ((x2 - x1) or 1)
7  for x in range(x1, x2 + 1):
8   y1 += m
9   set_pixel(x, round(y1), c)
```

Editor | Shell

$m \leftarrow (y_2-y_1)/(x_2-x_1)$
Pour tout entier $x \in [x_1,x_2]$:
  $y_1 \leftarrow y_1+m$
  allumer pixel (**x**,*arrondi*($\mathbf{y_1}$))

Pour optimiser :

- ➢ Remplaçons directement par **y=y+m**
- ➢ Au lieu de créer **y**, réutilisons $\mathbf{y_1}$
- ➢ Et faisons pareil pour **draw_yline()**

draw_line(032,000,000,176,...)

Temps de tracé sur calculatrice Graph Math+ :
42,07s → 35,57s → **27,79s** *(-34%)*

Optimisation draw_line “à la Bresenham”

Idée 3 : “d’après l’algorithme de Bresenham”

Algorithme par Jack Bresenham (IBM), 1962, ici dans une version généralisée à toutes les directions et optimisée

Avantages :

- pas de division
- ne travaille que sur des entiers

Le principe “à la tortue” :

- on part d’une extrémité du segment
- on note le sens horizontal **sx** et le sens vertical **sy** pour atteindre l’autre extrémité
- soit un entier relatif **err**, codant à la fois l’écart horizontal et vertical par rapport au segment théorique
- Tant que l’on n’a pas atteint l’autre extrémité :
  - Si l’écart horizontal est acceptable :
    - Déplacement de 1 pixel horizontal
    - Mise à jour de **err**
  - Si l’écart vertical est acceptable :
    - Déplacement de 1 pixel vertical
    - Mise à jour de **err**

Temps de tracé sur calculatrice Graph Math+ :

42,07s → … → 27,79s → **11,35s** *(-73%)*

dlin5.py

```
from casioplot import *

def sign(x):
  return x>0 and 1 or x<0 and -1

def draw_line(x1, y1, x2, y2, c):
  x1, y1, x2, y2 = [
    round(v)
    for v in (x1, y1, x2, y2)
  ]
  dx, dy = x2 - x1, y2 - y1
  sx, sy = sign(dx), sign(dy)
  dx, dy = abs(dx), abs(dy)
  err = dx - dy
  while True:
    set_pixel(x1, y1, c)
    if x1 == x2 and y1 == y2:
      break
    e2 = 2 * err
    if e2 > -dy:
      err -= dy
      x1 += sx
    if e2 < dx:
      err += dx
      y1 += sy
```

Editor | Shell

draw_line(032,000,000,176,...)

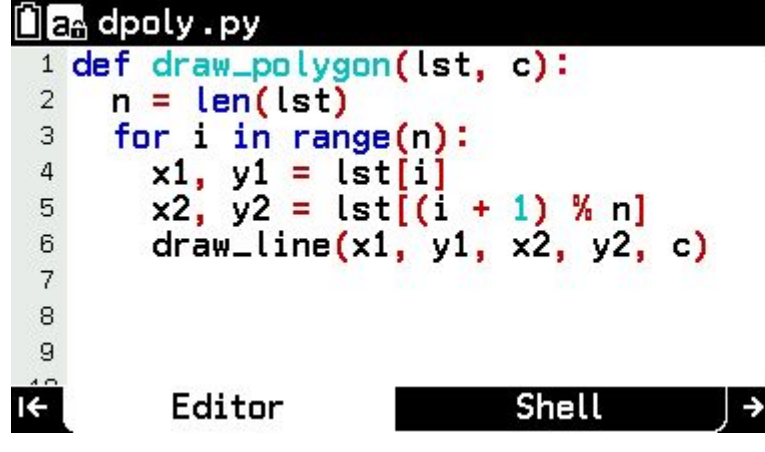
Extension de casioplot avec polygones

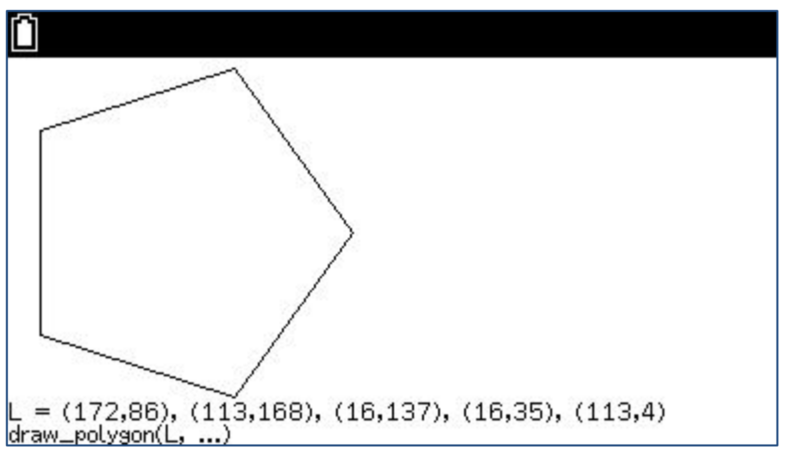
Étendons maintenant la bibliothèque **casioplot** avec une fonction **draw_polygon()**.

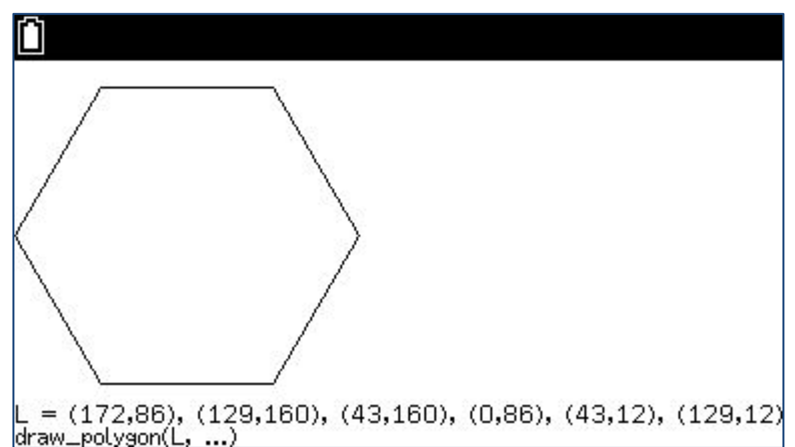
Spécifications : draw_polygon(lst, c), avec :

- **lst** : liste des coordonnées des sommets
- **c** : la couleur du trait

Principe : Tracer les segments pour toutes les paires de sommets consécutifs.







Mise en place d’une perspective 3D

La scène en 3D contient :

- Les cubes (de face)
- La vaisseau (triangle au sol)
- Un observateur

Le champ de vision est une **pyramide**.

Le rendu 3D consiste à **projeter** la pyramide sur un écran proche de l’observateur.

On choisit les axes :

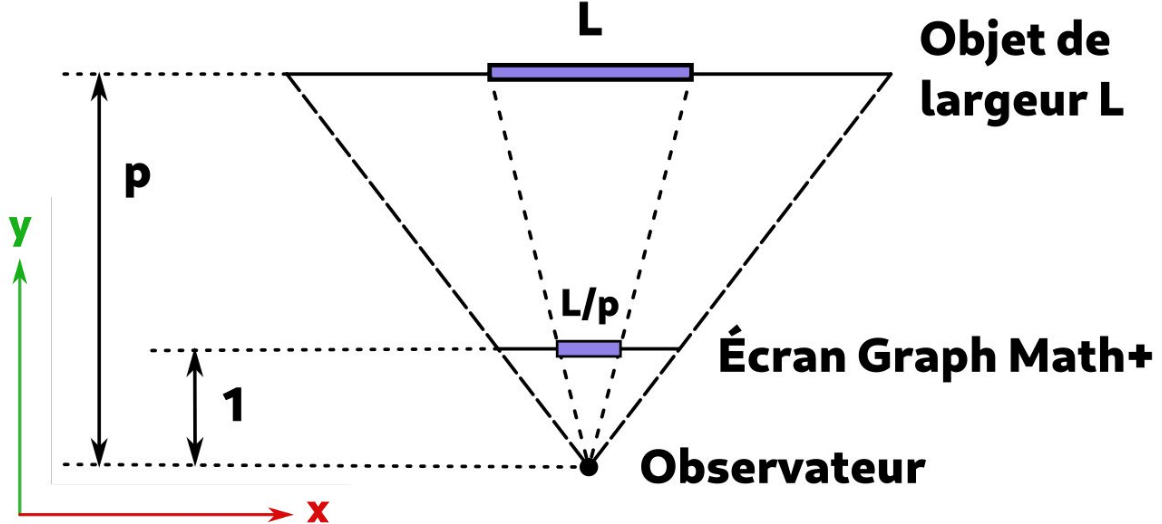
- (Ox) vers la droite de l’écran
- (Oy) vers le fond de la scène
- (Oz) vers le haut de l’écran

y
z
x
Fin de l'espace
Écran Graph Math+
Observateur

Principe du calcul de perspective

Projeter un objet de la profondeur p vers l'écran divise sa taille par p.

Pourquoi ? Théorème de Thalès !



Intuition pour la perspective : $(x, y, z) \rightarrow (x/y, z/y)$

Principe du calcul de perspective

Prenons donc comme coordonnées de caméra :

- x_camera=0 (au niveau de la médiane verticale de l'écran)
- y_camera=0 initialement (à augmenter avec l'avancement dans le jeu)
- z_camera=2.5

Pour les 3 sommets du vaisseau triangulaire, choisissons comme coordonnées :

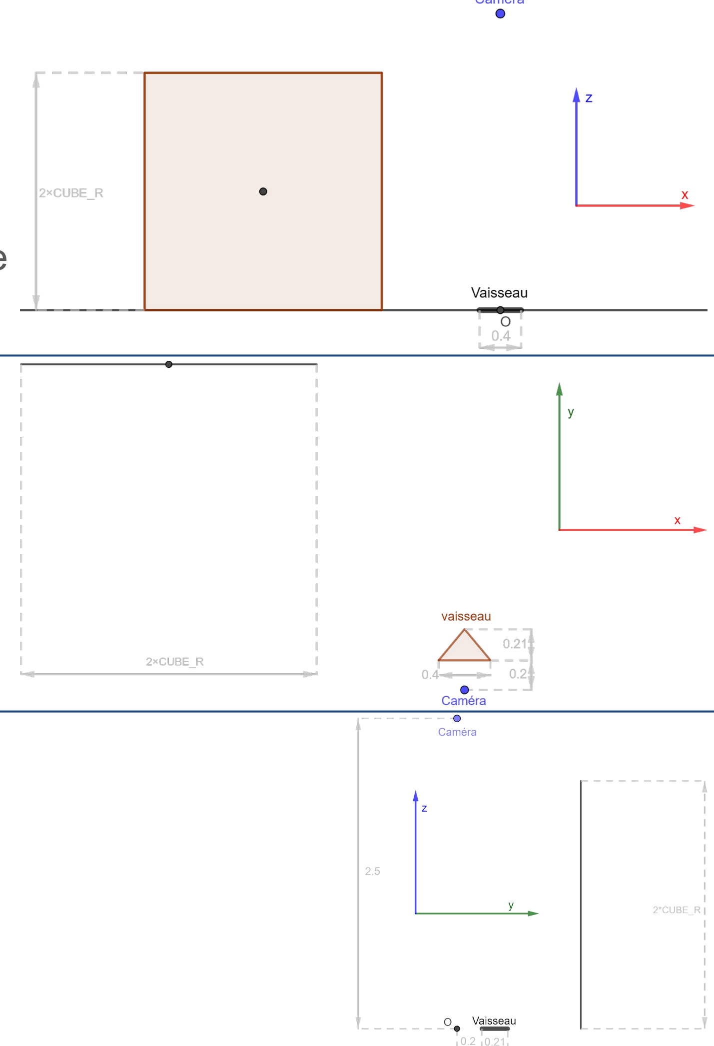
- (x_camera - 0.2, y_camera + 0.2, 0)
- (x_camera + 0.2, y_camera + 0.2, 0)
- (x_camera, y_camera + 0.21, 0)

Pour caractériser nos carrés dans l'espace 3D, choisissons :

- Un côté de longueur 2 (demi-côté CUBE_R=1)
- Le centre de coordonnées (x, y, z) avec z=CUBE_R

Les 4 sommets à dessiner ont alors pour coordonnées :

- (x + CUBE_R, y, 2*CUBE_R)
- (x - CUBE_R, y, 2*CUBE_R)
- (x + CUBE_R, y, 0)
- (x - CUBE_R, y, 0)

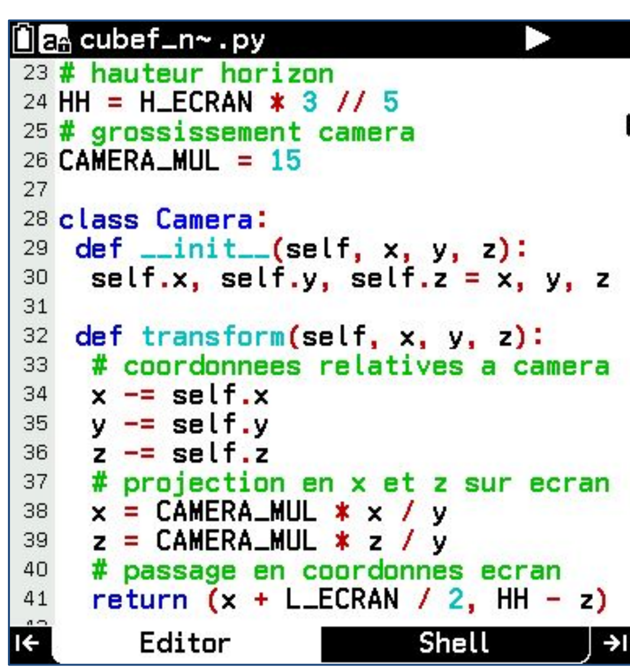


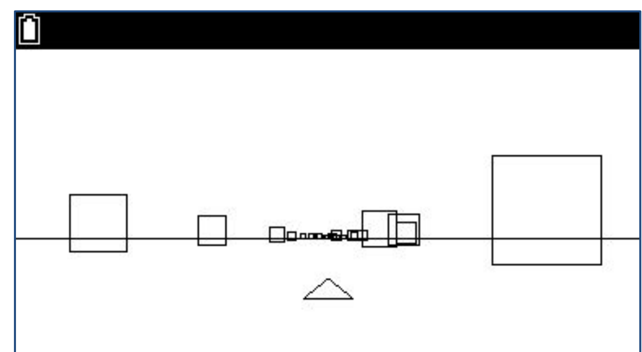
Implémentation !

Calculons la position où le point (x, y, z) arrive à l'écran.

1. Calculer la position *par rapport à l'observateur*
   - $x -= x_{obs}$, $y -= y_{obs}$, $z -= z_{obs}$
2. Faire le calcul de projection
   - $x = x/y$
   - $z = z/y$
3. (Prochain slide !)
   Changement de repère pour passer en pixels **casioplot**.

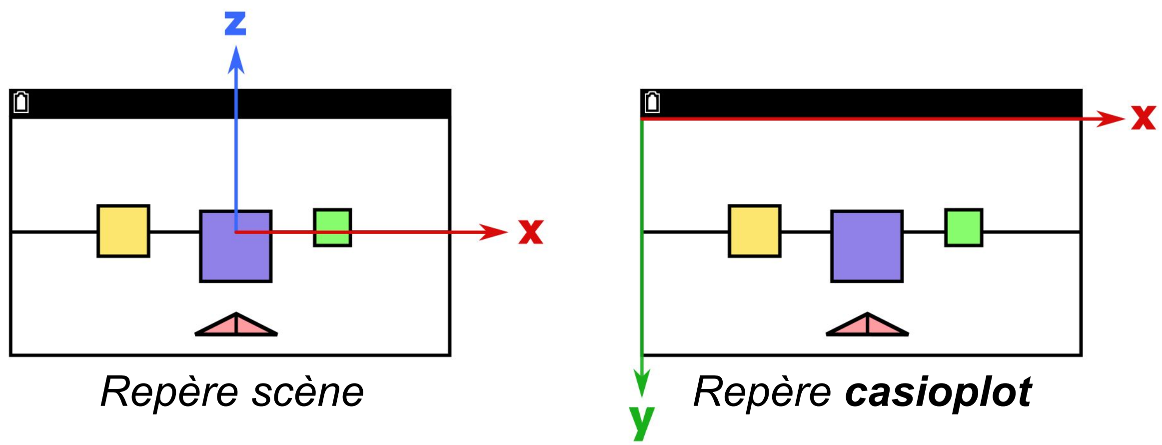
Et voilà ! 🎉





Dernier ajustement pour le dessin

Après projection, on a un repère (x, z) qui mesure en mètres et est centré sur l'horizon.
Mais l'écran mesure en pixels et commence en haut à gauche de l'écran.



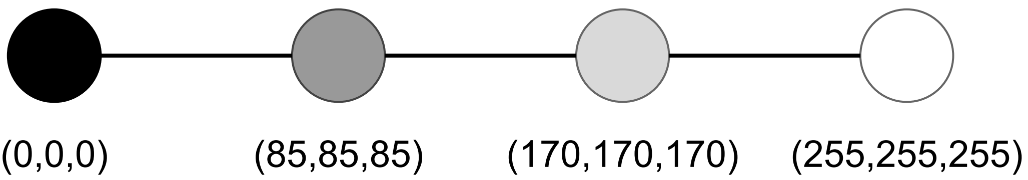
La transformation est une fonction affine (admise) :

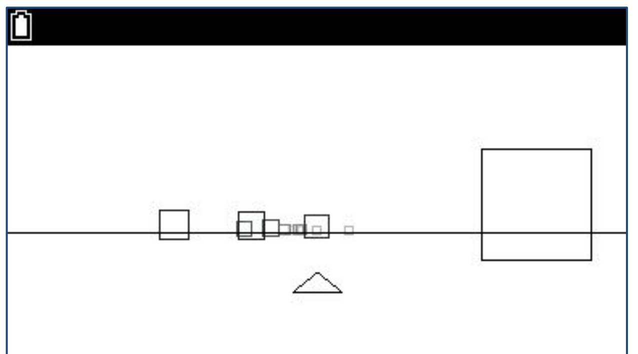
- $x_{écran}$ = ⟨pixels par mètre⟩ × $x_{projection}$ – ⟨largeur écran⟩ / 2
- $y_{écran}$ = ⟨hauteur horizon⟩ – ⟨pixels par mètre⟩ × $z_{projection}$

Effet de profondeur par couleur

La visibilité est pas top parce que les carrés s'entassent.

Effet « brouillard » : dessiner les carrés du fond en gris.





gray_level = ⟨proche de 0 si y est petit, de 255 si y est grand⟩
color = (gray_level, gray_level, gray_level)

Ordre d'affichage :

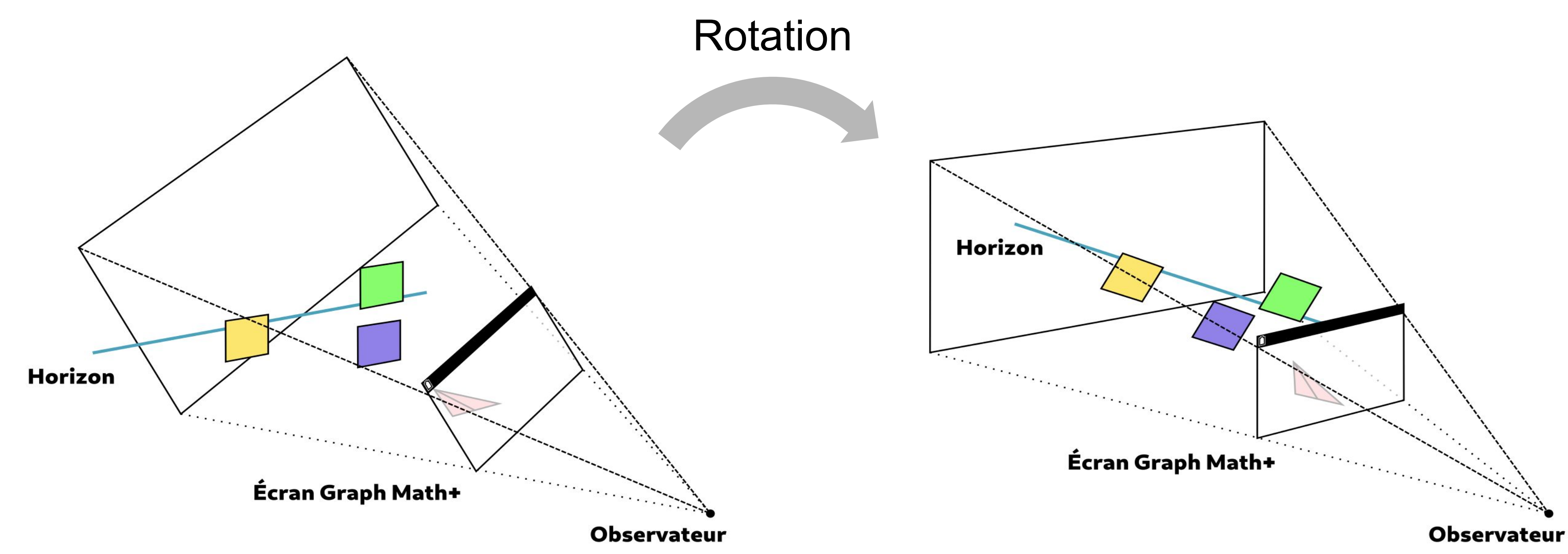
- Les objets du fond d'abord ⚠️
- Ici, on triche : horizon en dernier (meilleure visibilité)

```
cubef_n~.py
80 def render(x, y, vx, vy, cubes):
81     camera = Camera(x, y, 0.5)
82     clear_screen()
83     render_cubes(cubes, camera)
84     render_player(camera)
85     render_horizon(camera)
86
87
88
```

Editor   Shell

Déplacements et rotation #1

Au lieu de faire des calculs compliqués quand l'observateur est penché…
on peut **se ramener au cas facile** en faisant tourner toute la scène.



Déplacements et rotation #2

💡 Idée importante : **l’observateur est le centre du monde.**

- On fait tourner la scène pour que l’observateur soit droit
- On translate la scène pour que l’observateur soit à l’origine

Et donc on continue d’utiliser les mêmes formules. 😀

🛠️ Optimisations sur le rendu 3D dans `cubef.py` :

- On projette un seul point par carré
- On en déduit les deux autres en tenant compte de la rotation

Simulation physique

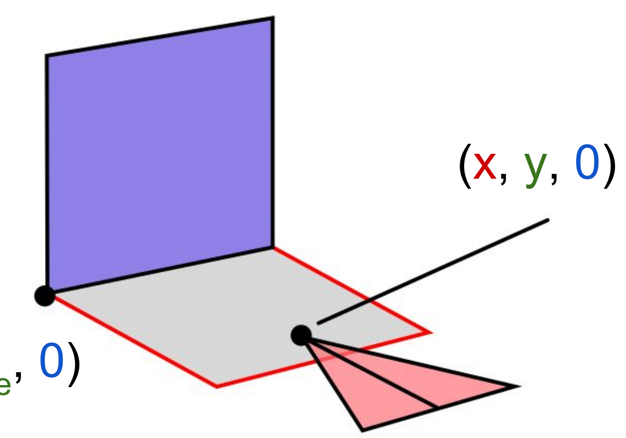
Déplacement des objets :

- On fait avancer l'observateur : y_{obs} += ⟨vitesse vaisseau⟩
- Si on appuie à droite : x_{obs} += ⟨vitesse latérale⟩

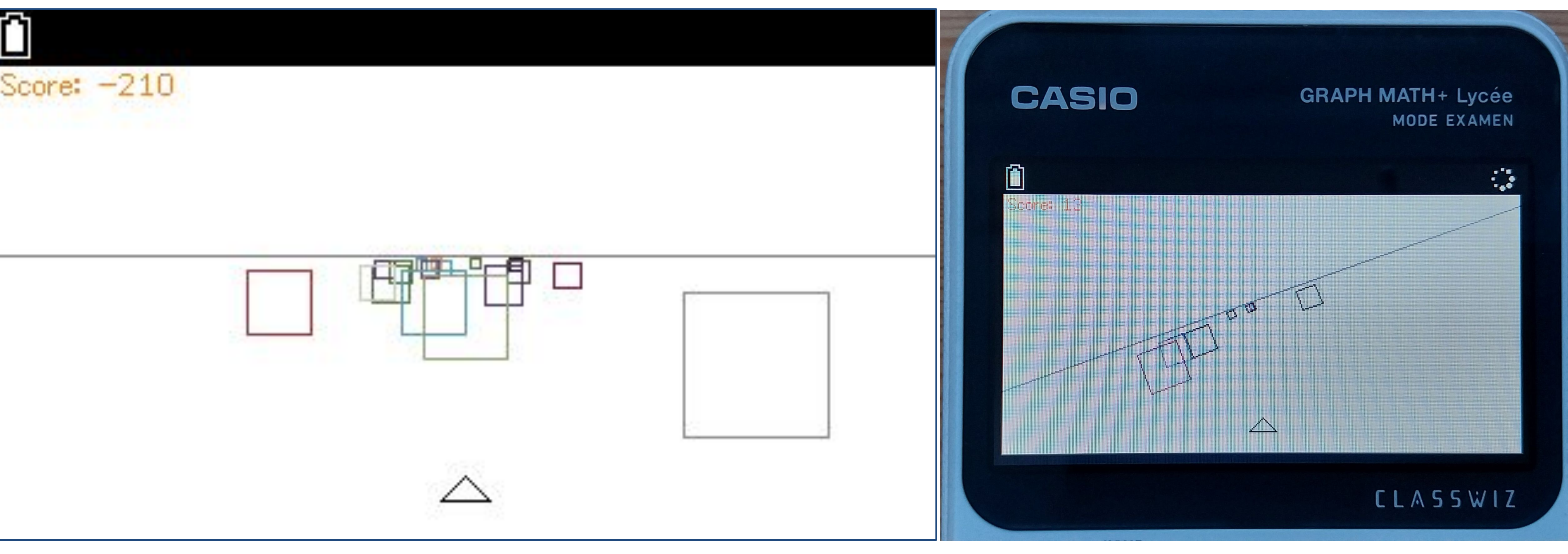
  … à gauche : x_{obs} –= ⟨vitesse latérale⟩

Test si le joueur est dans la « zone de danger » d'un cube :

- $x_{cube} \le x \le x_{cube}$ + ⟨largeur cube⟩ et
- y_{cube} – ⟨taille zone de danger⟩ $\le y \le y_{cube}$



CubeField pour Casio Graph Math+



Merci pour votre attention !

Téléchargement du jeu, des exemples et des ressources :
https://www.casio-education.fr/contenus/creer-un-jeu-video-avec-les-calculatrices-graphiques

Téléchargement autres jeux Python évoqués :

<table>
<tr><th>Graph 35+E II</th><th>Graph 90+E</th><th>Graph Math+</th></tr>
<tr><td>Flappy Bird
1000 Bornes
Pykaster 3D</td><td>Flappy Bird
1000 Bornes
Pykaster 3D
Saute Mouton
Etiord City
Gravity Guy</td><td rowspan="2">Orlog
WHIS
Boules Bleues
6 Qui Prend
Jeu de Mémoire
Jeu de mains…
Couleurs RGB</td></tr>
<tr><td colspan="2">Synchro Donjon
Alrys</td></tr>
</table>

Clé USB d’émulation Casio :

- ➢ Emulateur Graph Math+
- ➢ Demande clé USB gratuite

Émulateur en ligne Graph Math+ :
ClassPad Academy
https://classpad.academy/fr

Animé par :

- Planète Casio - https://www.planet-casio.com
- TI-Planet - https://tiplanet.org