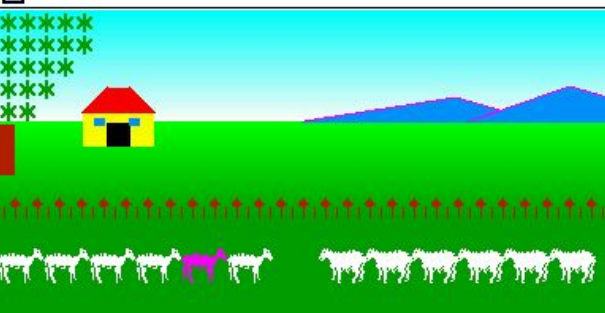


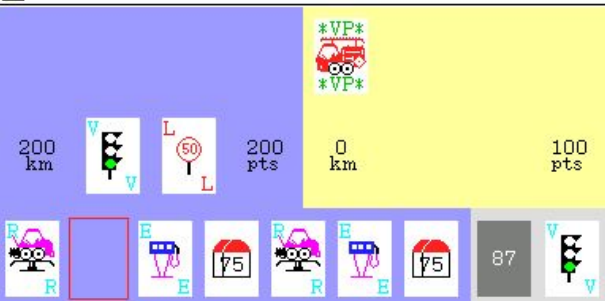
📄 **Saute-Moutons** (rentrée 2020)



📄 **RPG Alrys** (concours rentrée 2021)



📄 **1000 Bornes** (concours 2023-2024)



# Atelier jeu vidéo

Quelques-uns de nos jeux  
Python pour **Graph 90+E** et  
**Graph 35+E II** :

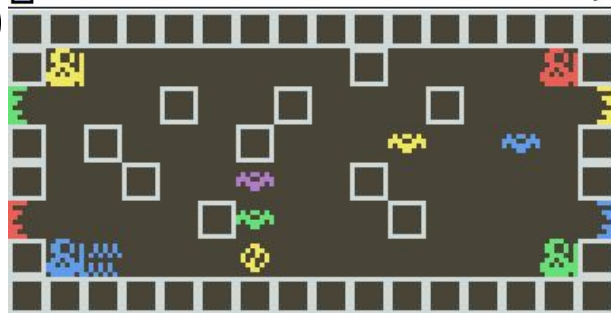
📄 **Doom-like Pykaster3D** (concours 2022)



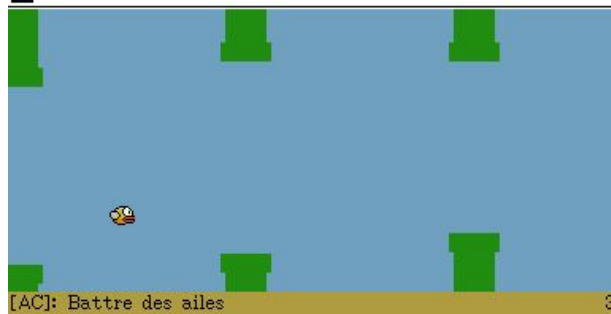
📄 **Etiord City** (atelier 1er trimestre 2023)



📄 **SynchroD** (concours rentrée 2021)



📄 **FlappyBird** (atelier 1er trimestre 2022)

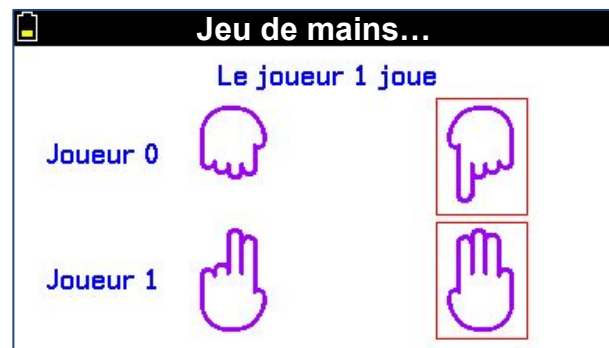
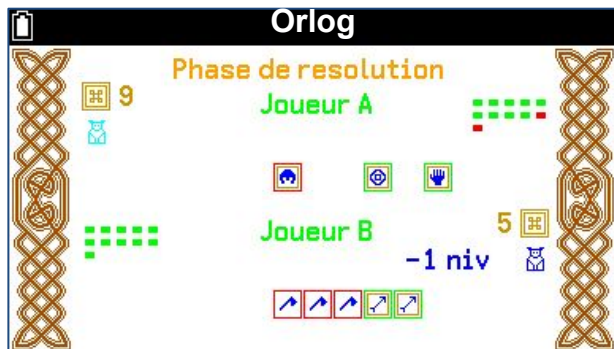


📄 **Gravity Guy** (atelier 1er trimestre 2023)



## 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

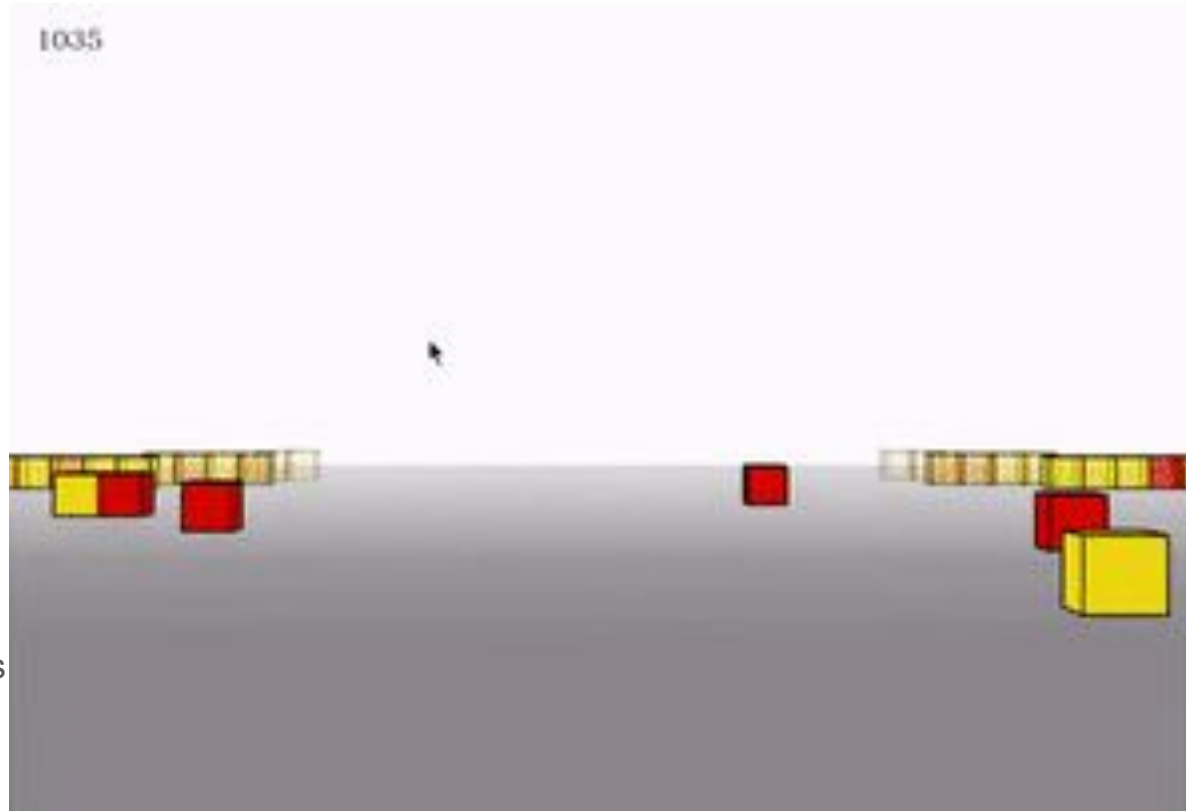
**Objectifs** :

- 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 !

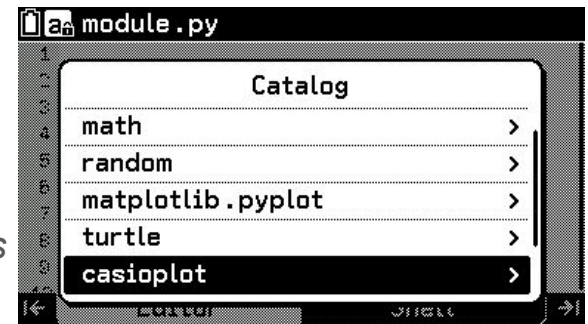
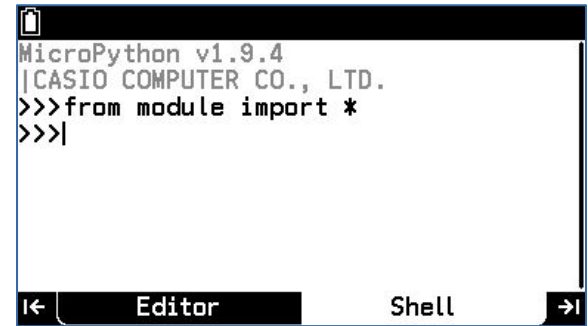
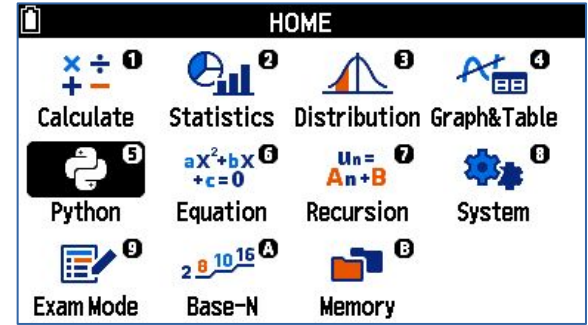


# L'application Python Graph Math+

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

5 bibliothèques intégrées :

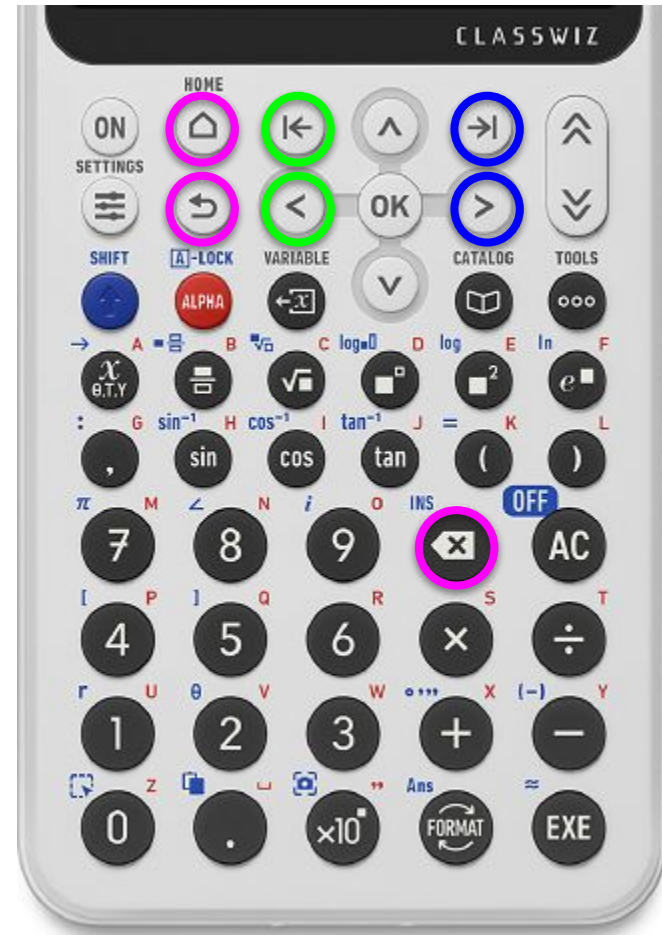
- 3 bibliothèques standard **Python 3** :
  - **math**  
*fonctions mathématiques*
  - **random**  
*génération de données aléatoires*
  - **turtle**  
*tracé par déplacements (à la Scratch / Logo)*
- 1 bibliothèque tierce populaire :
  - **matplotlib** (limitée à **matplotlib.pyplot**)  
*tracé de diagrammes 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 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
```

Editor Shell

Catégorie	Touche	Code
directions		23
		25
		14
		34
		13
		15
		16
		26
validation actions		95
		24
		31
		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

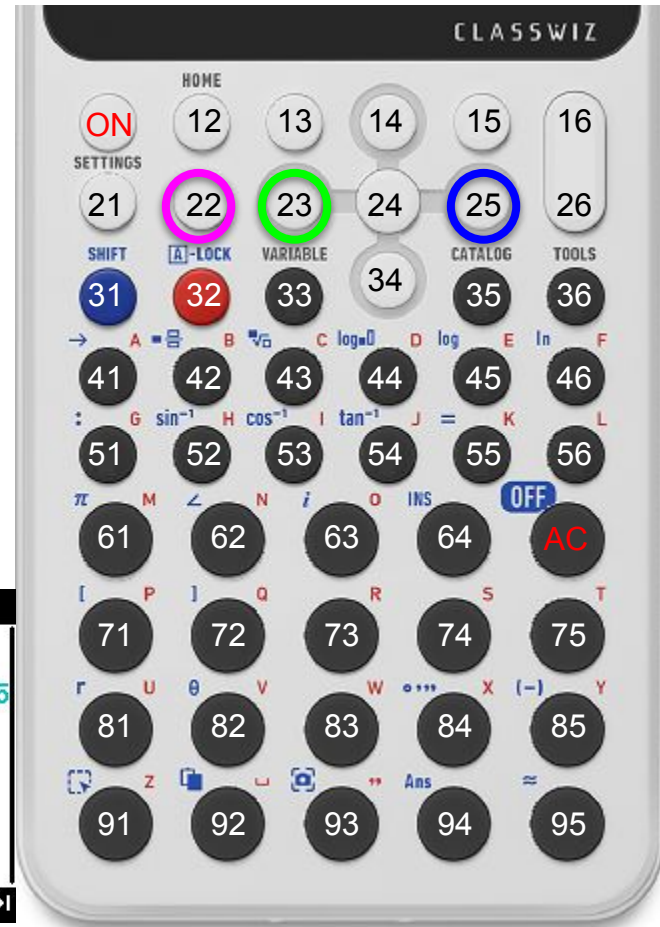
Retenons et définissons :

- **K\_ESC = 22**
- **K\_LEFT = 23**
- **K\_RIGHT = 25**

Boucle principale du jeu :

```
module.py
1 from casioplot import *
2
3 K_ESC, K_LEFT, K_RIGHT = 22, 23, 25
4
5 key = None
6 while key != K_ESC:
7     #...
8     key = getkey()
9
10
```

Editor Shell



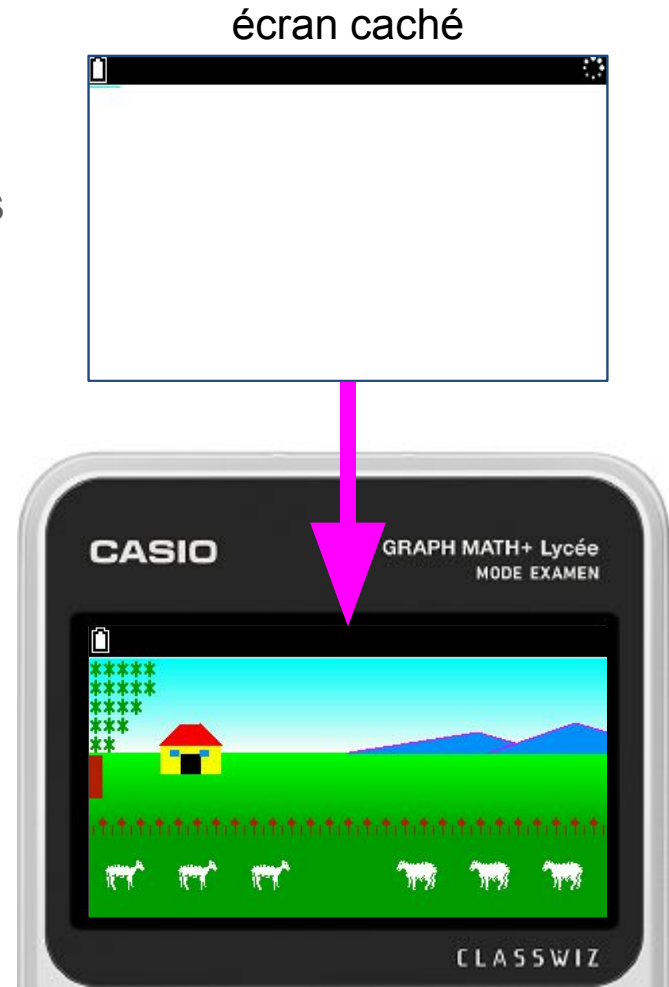
# Casioplot et principe d'affichage

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*

```
module .py
1
2
3      Catalog > casioplot
4      casioplot .
5      clear_screen()
6      draw_string(,,)
7
8      from casioplot import *
9      getKey()
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
module .py
1
2
3      Catalog > casioplot
4      getKey()
5      get_pixel(,)
6      import casioplot
7      set_pixel(,)
8      show_screen()
```

# Casioplot et zone graphique

Plusieurs zones écran réservées non modifiables :

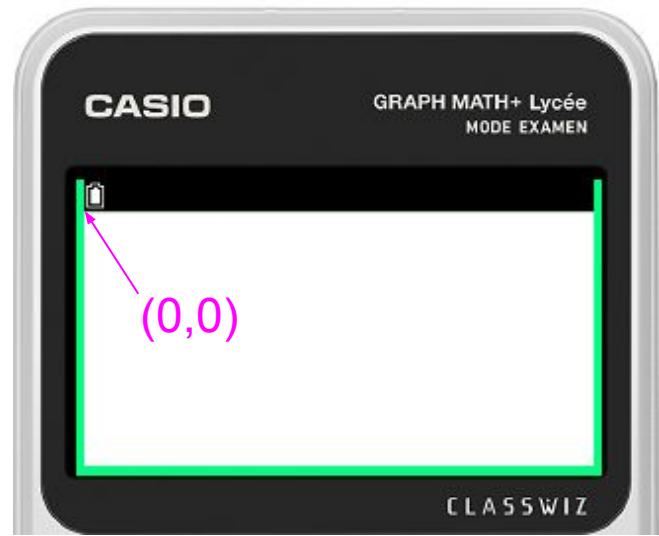
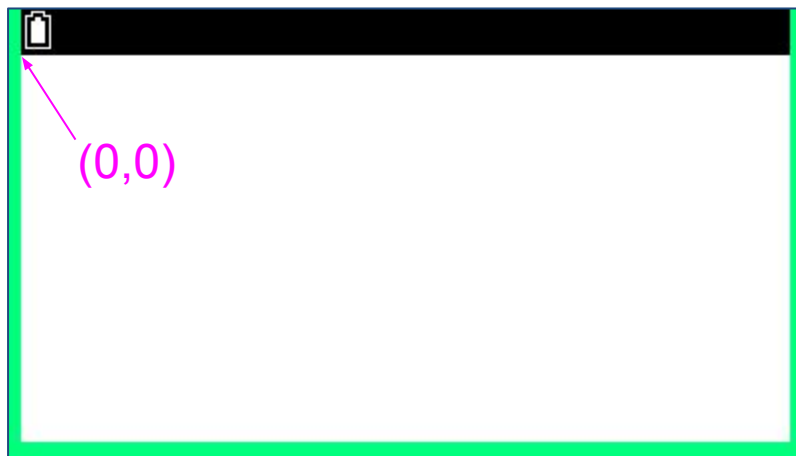
- En haut, barre de statut de 24 pixels de haut
- Marges du mode examen, en noir ou vert :
  - 6 pixels de large à gauche et à droite
  - 8 pixels de haut en bas

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
1 def scr_size():
2     x, y, dx, dy = 0, 0, 1, 1
3     while dx or dy:
4         set_pixel(x, y, (255, 0, 0))
5         show_screen()
6         x, y = x + dx, y + dy
7         if get_pixel(x, y) == None:
8             if dx and get_pixel(x-1, y):
9                 x, dx = x - 1, 0
10            if dy and get_pixel(x, y-1):
11                y, dy = y - 1, 0
12    return x + 1, y + 1
```

Editor Shell

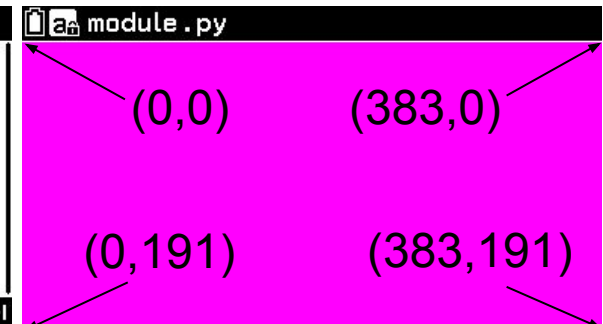
(8, 8)

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)

```
scr_fill.py
1 from casioplot import *
2
3 L, H = 384, 192
4 c = (255, 0, 255)
5 for y in range(H):
6     for x in range(L):
7         set_pixel(x, y, c)
8 show_screen()
```

Editor Shell



# 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

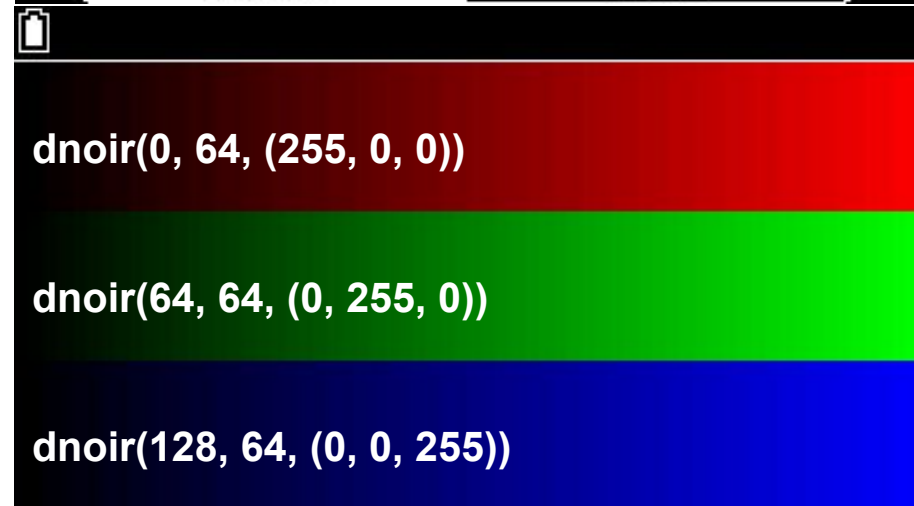
L'écran **Graph Math+** affiche les couleurs les plus proches sur l'échelle **RVB 565**, codant :

- 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)

```
>>>set_pixel(0, 0, (7,7,7))
>>>get_pixel(0, 0)
(0, 4, 0)
>>>set_pixel(0, 0, (13,13,13))
>>>get_pixel(0, 0)
(8, 12, 8)
>>>set_pixel(0, 0, (17,17,17))
>>>get_pixel(0, 0)
(16, 16, 16)
```

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()
```



# Choix graphiques pour Cubefield

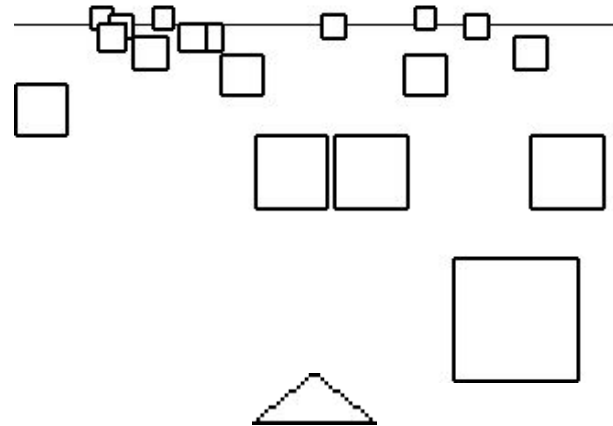
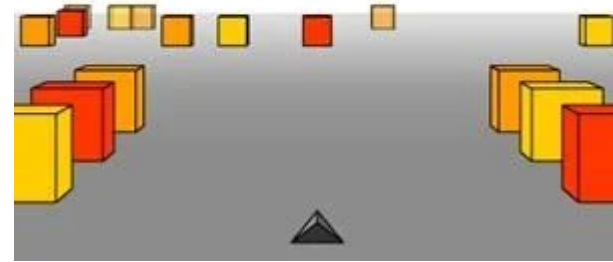
Tout le dessin (sauf texte) passe par **set\_pixel()**.

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 casioplote avec draw\_line #1

Étendons `casioplote` avec une fonction pour tracer un segment entre 2 points.

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

- $x_1, y_1$  : coordonnées d'une des extrémités du segment
- $x_2, y_2$  : coordonnées de l'autre extrémité du segment
- $c$  : la couleur du trait

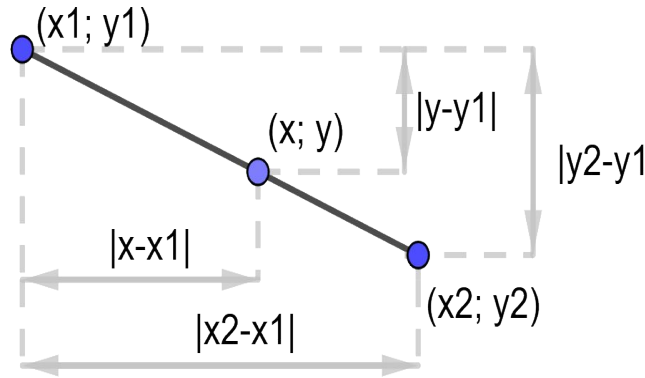
## *Idée 1 : "approche naïve"*

Allumer tous les points d'abscisse entière du segment.

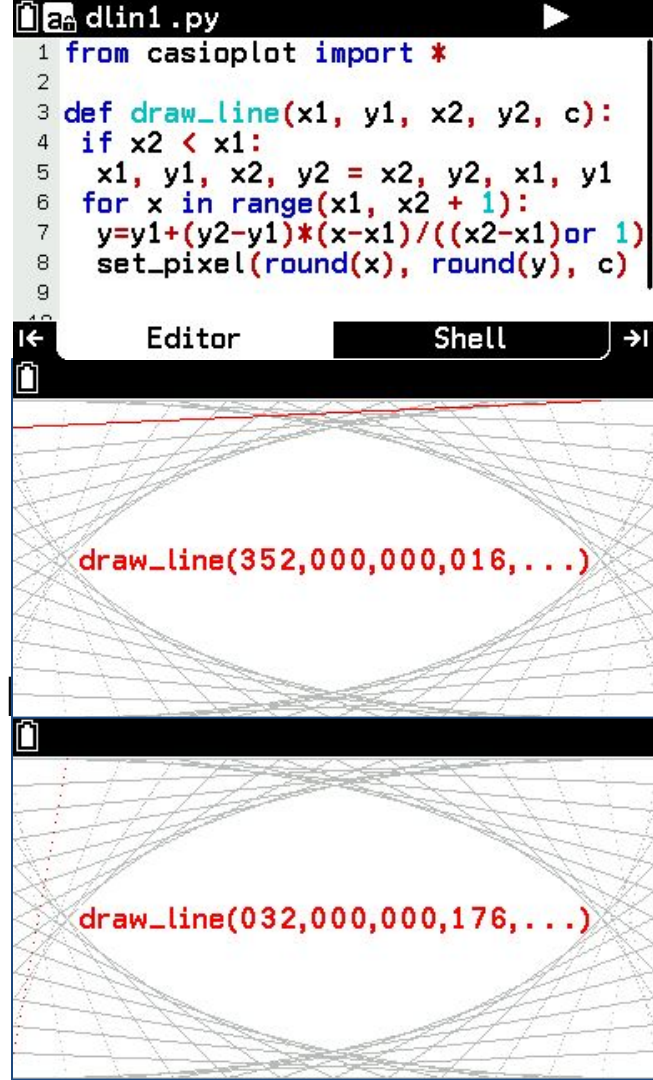
Pour tout entier  $x \in [x_1, x_2]$  :

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

allumer le pixel  $(x, y)$



**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  $y \in [y_1, y_2]$  :

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

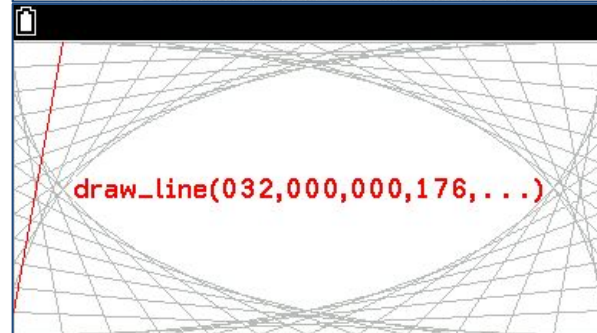
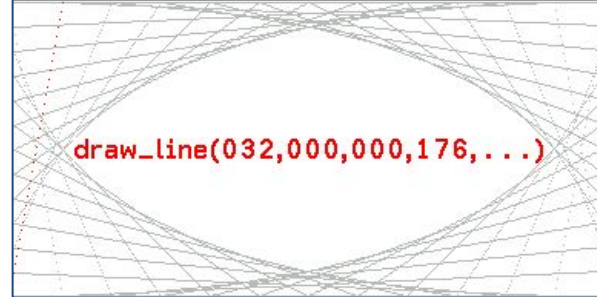
allumer le pixel (*arrondi(x), y*)

- 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**

```
dlin2.py
10 def draw_yline(x1, y1, x2, y2, c):
11     if y2 < y1:
12         x1, y1, x2, y2 = x2, y2, x1, y1
13     for y in range(y1, y2 + 1):
14         x = x1 + (x2 - x1) * (y - y1) / ((y2 - y1) or 1)
15         set_pixel(round(x), round(y), c)
16
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
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Editor Shell



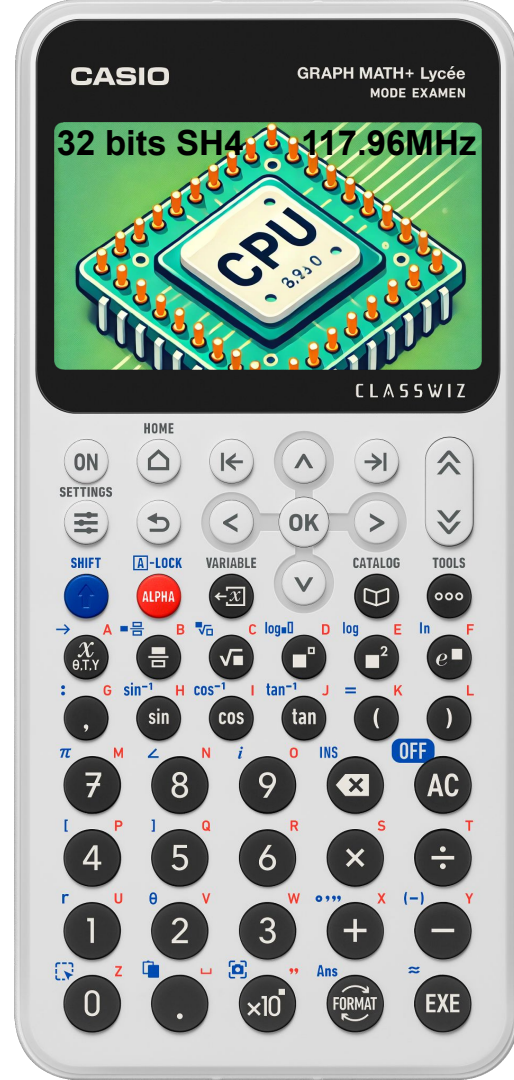
# À la recherche du temps perdu...

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()` :

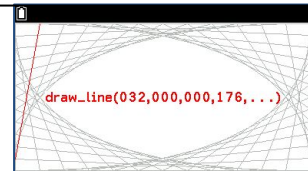
- Nous calculons  $y=y_1+(y_2-y_1)\times(x-x_1)/(x_2-x_1)$
- Alors que  $(y_2-y_1)/(x_2-x_1)$  est constant
- Et nous arrondissons  $x$

Pour optimiser :

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

<pre>dlin2.py 1 from casioplot import * 2 3 def draw_xline(x1, y1, x2, y2, c): 4     if x2 &lt; 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 10 def draw_yline(x1, y1, x2, y2, c): 11     if x2 &lt; x1: 12         x1, y1, x2, y2 = x2, y2, x1, y1 13     for y in range(y1, y2 + 1): 14         x=y1+(x2-y1)*(y-y1)/((y2-y1)or 1) 15         set_pixel(round(x), round(y), c) 16 17 def draw_line(x1, y1, x2, y2, c): 18     if abs(y2 - y1) &gt; 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 26 27</pre>	<pre>dlin3.py 1 from casioplot import * 2 3 def draw_xline(x1, y1, x2, y2, c): 4     if x2 &lt; 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) 10 11 def draw_yline(x1, y1, x2, y2, c): 12     if x2 &lt; x1: 13         x1, y1, x2, y2 = x2, y2, x1, y1 14     m = (x2 - x1) / ((y2 - y1) or 1) 15     for y in range(y1, y2 + 1): 16         x = x1 + (y - y1)*m 17         set_pixel(round(x), y, c) 18 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) &gt; abs(x2 - x1): 25         draw_yline(x1, y1, x2, y2, c) 26     else: 27         draw_xline(x1, y1, x2, y2, c) 28</pre>
<p>Pour tout entier <math>x \in [x_1, x_2]</math> :</p> $y \leftarrow y_1 + (y_2 - y_1) \times (x - x_1) / (x_2 - x_1)$ <p>allumer pixel (<math>x</math>, <i>arrondi</i>(<math>y</math>))</p>	<p><math>m \leftarrow (y_2 - y_1) / (x_2 - x_1)</math></p> <p>Pour tout entier <math>x \in [x_1, x_2]</math> :</p> $y \leftarrow y_1 + (x - x_1) \times m$ <p>allumer pixel (<math>x</math>, <i>arrondi</i>(<math>y</math>))</p>

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



# Optimisation draw\_line #2

## Idée 2 : "supprimer la multiplication"

Par exemple dans la boucle `draw_xline()`

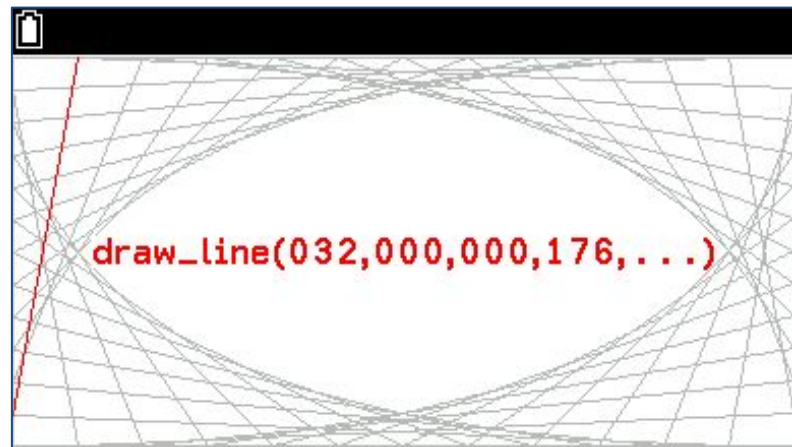
- Nous calculons  $y = y_1 + (x - x_1) \times m$
- À chaque itération,  $x$  augmente de 1
- Donc  $y$  augmente de  $m$

<pre>1 from casioplot import * 2 3 def draw_xline(x1, y1, x2, y2, c): 4     if x2 &lt; 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)</pre>	<pre>1 from casioplot import * 2 3 def draw_xline(x1, y1, x2, y2, c): 4     if x2 &lt; 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)</pre>
<p><math>m \leftarrow (y_2 - y_1) / (x_2 - x_1)</math> Pour tout entier <math>x \in [x_1, x_2]</math> :</p> <p><math>y \leftarrow y_1 + (x - x_1) \times m</math> allumer pixel <math>(x, \text{arrondi}(y))</math></p>	<p><math>m \leftarrow (y_2 - y_1) / (x_2 - x_1)</math> Pour tout entier <math>x \in [x_1, x_2]</math> :</p> <p><math>y_1 \leftarrow y_1 + m</math> allumer pixel <math>(x, \text{arrondi}(y_1))</math></p>

Pour optimiser :

- Remplaçons directement par  $y = y + m$
- Au lieu de créer  $y$ , réutilisons  $y_1$
- Et faisons pareil pour `draw_yline()`

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



# Optimisation draw\_line “à la Bresenham”

## *Idee 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 :

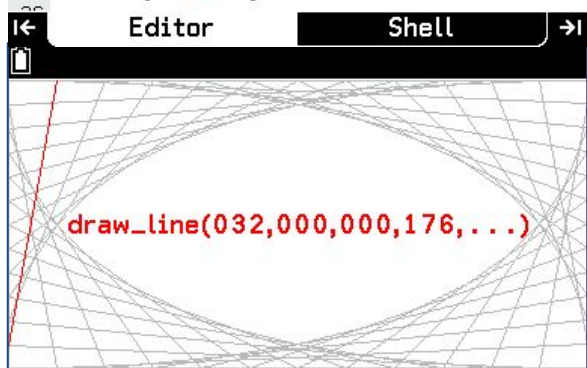
- pas de division
- ne travaille que sur des entiers

### 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
1 from casioplot import *
2
3 def sign(x):
4     return x>0 and 1 or x<0 and -1
5
6 def draw_line(x1, y1, x2, y2, c):
7     x1, y1, x2, y2 = [
8         round(v)
9         for v in (x1, y1, x2, y2)
10    ]
11    dx, dy = x2 - x1, y2 - y1
12    sx, sy = sign(dx), sign(dy)
13    dx, dy = abs(dx), abs(dy)
14    err = dx - dy
15    while True:
16        set_pixel(x1, y1, c)
17        if x1 == x2 and y1 == y2:
18            break
19        e2 = 2 * err
20        if e2 > -dy:
21            err -= dy
22            x1 += sx
23        if e2 < dx:
24            err += dx
25            y1 += sy
26
```



# Extension de casioplott avec polygones

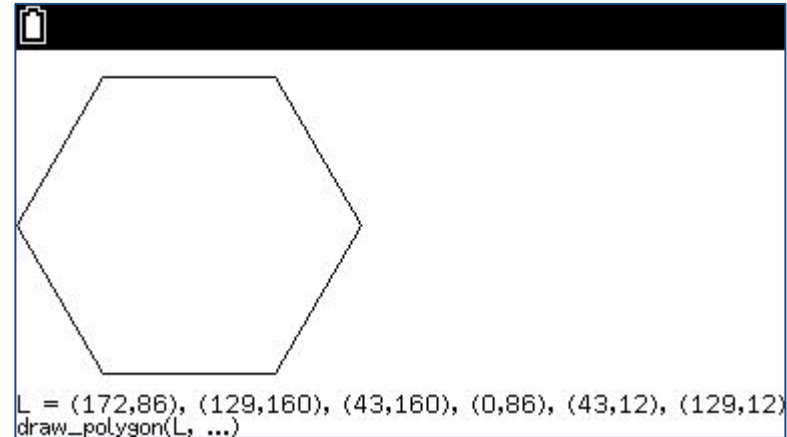
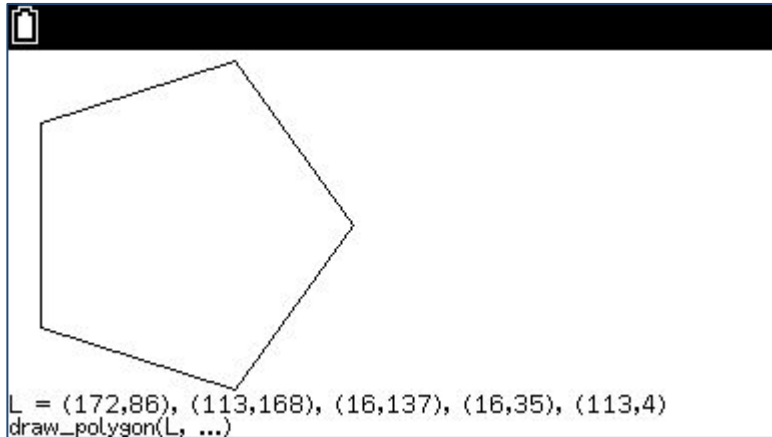
Étendons maintenant la bibliothèque **casioplott** avec une fonction **draw\_polygon()**.

**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.

```
dpoly.py
1 def draw_polygon(lst, c):
2     n = len(lst)
3     for i in range(n):
4         x1, y1 = lst[i]
5         x2, y2 = lst[(i + 1) % n]
6         draw_line(x1, y1, x2, y2, c)
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```



# Mise en place d'une perspective 3D

La scène en 3D contient :

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

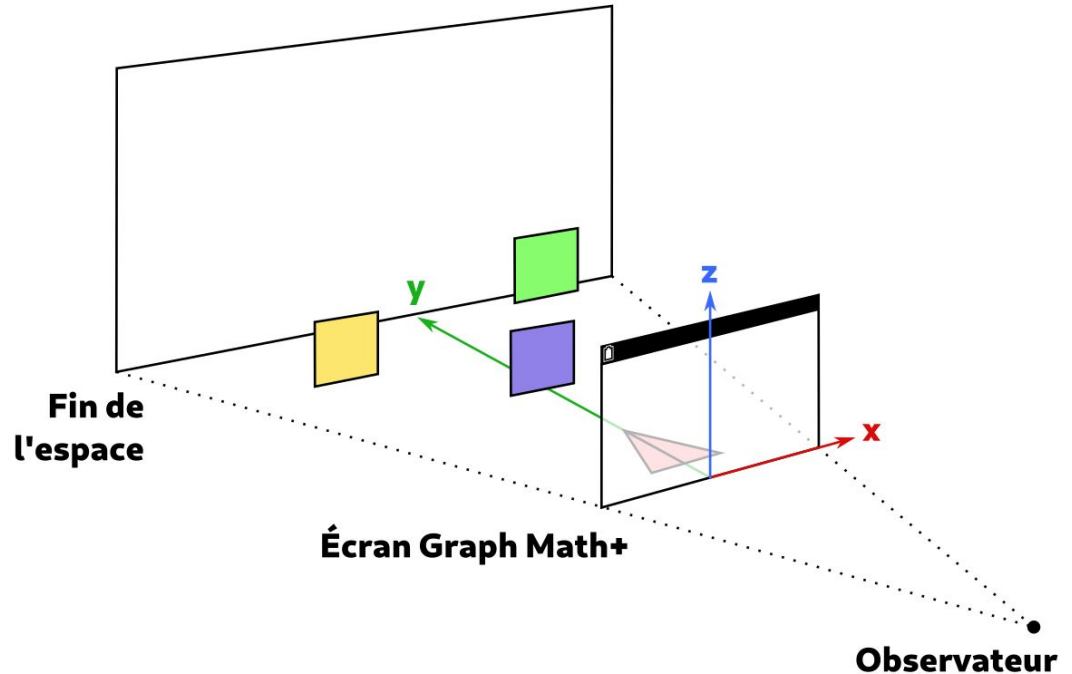
Le champ de vision est une **pyramide**.



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

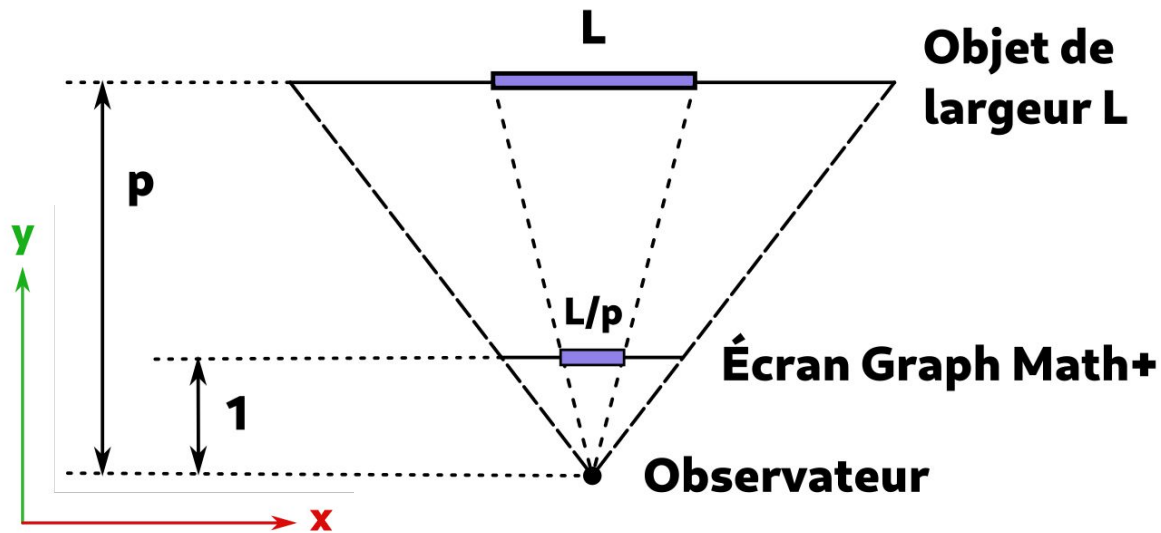
On choisit les axes :

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



# 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 :

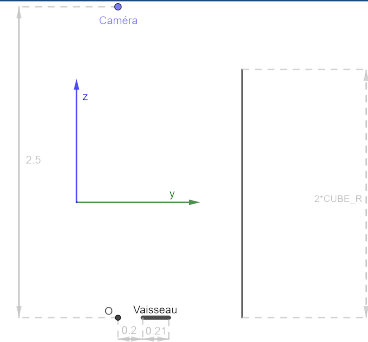
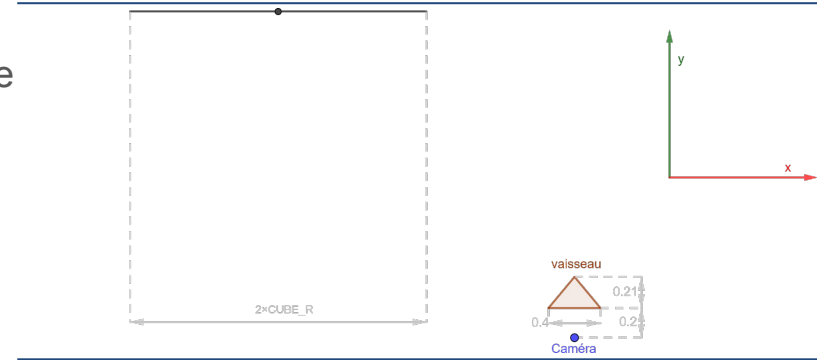
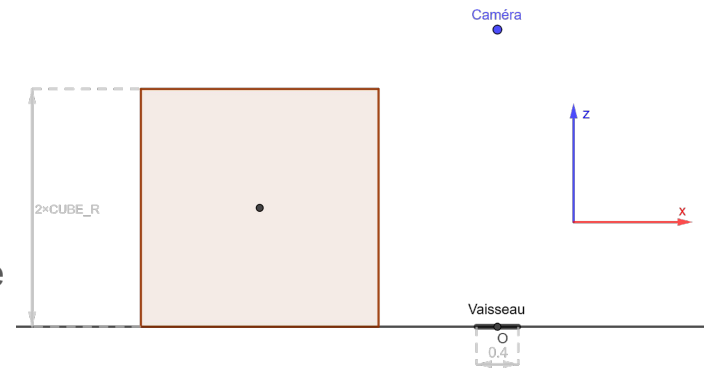
- $(x_{camera} - 0.2, y_{camera} + 0.2, 0)$
- $(x_{camera} + 0.2, y_{camera} + 0.2, 0)$
- $(x_{camera}, y_{camera} + 0.21, 0)$

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*

- $x \text{ --} x_{\text{obs}}, y \text{ --} y_{\text{obs}}, z \text{ --} z_{\text{obs}}$

2. Faire le calcul de projection

- $x = x/y$

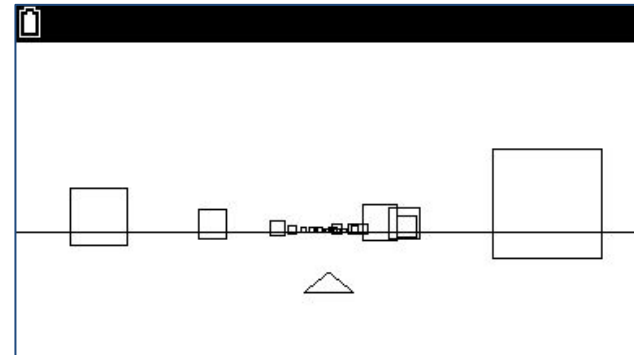
- $z = z/y$

3. (Prochain slide !)

Changement de repère pour passer en pixels **casioplot**.

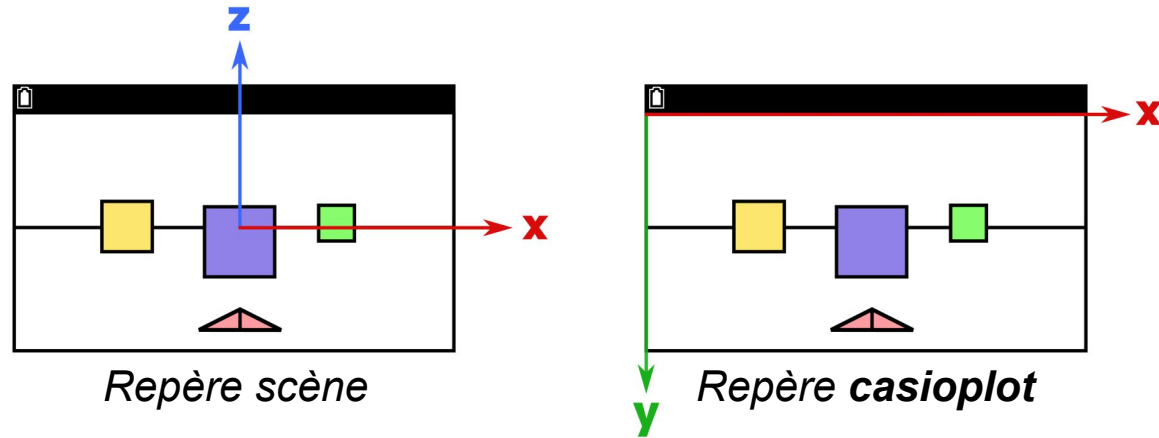
Et voilà ! 🎉

```
cubef_n~.py
23 # hauteur horizon
24 HH = H_ECRAN * 3 // 5
25 # grossissement camera
26 CAMERA_MUL = 15
27
28 class Camera:
29     def __init__(self, x, y, z):
30         self.x, self.y, self.z = x, y, z
31
32     def transform(self, x, y, z):
33         # coordonnees relatives a camera
34         x -= self.x
35         y -= self.y
36         z -= self.z
37         # projection en x et z sur ecran
38         x = CAMERA_MUL * x / y
39         z = CAMERA_MUL * z / y
40         # passage en coordonnes ecran
41         return (x + L_ECRAN / 2, HH - z)
```



# 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.



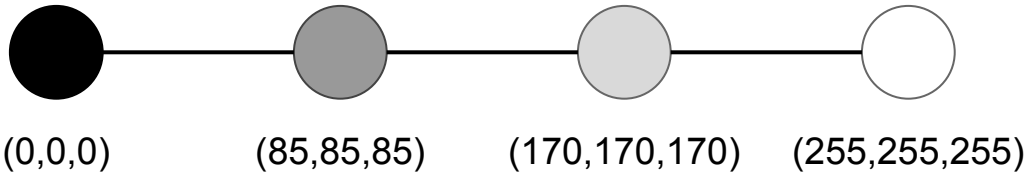
La transformation est une fonction affine (admise) :

- $x_{\text{écran}} = \langle \text{pixels par mètre} \rangle \times x_{\text{projection}} - \langle \text{largeur écran} \rangle / 2$
- $y_{\text{écran}} = \langle \text{hauteur horizon} \rangle - \langle \text{pixels par mètre} \rangle \times z_{\text{projection}}$

# Effet de profondeur par couleur

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

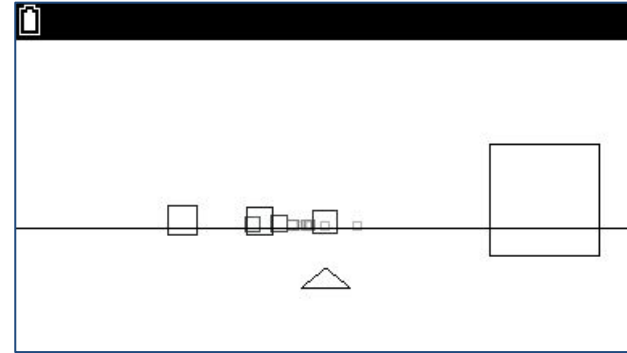
💡 Effet « brouillard » : dessiner les carrés du fond en gris.



```
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 :

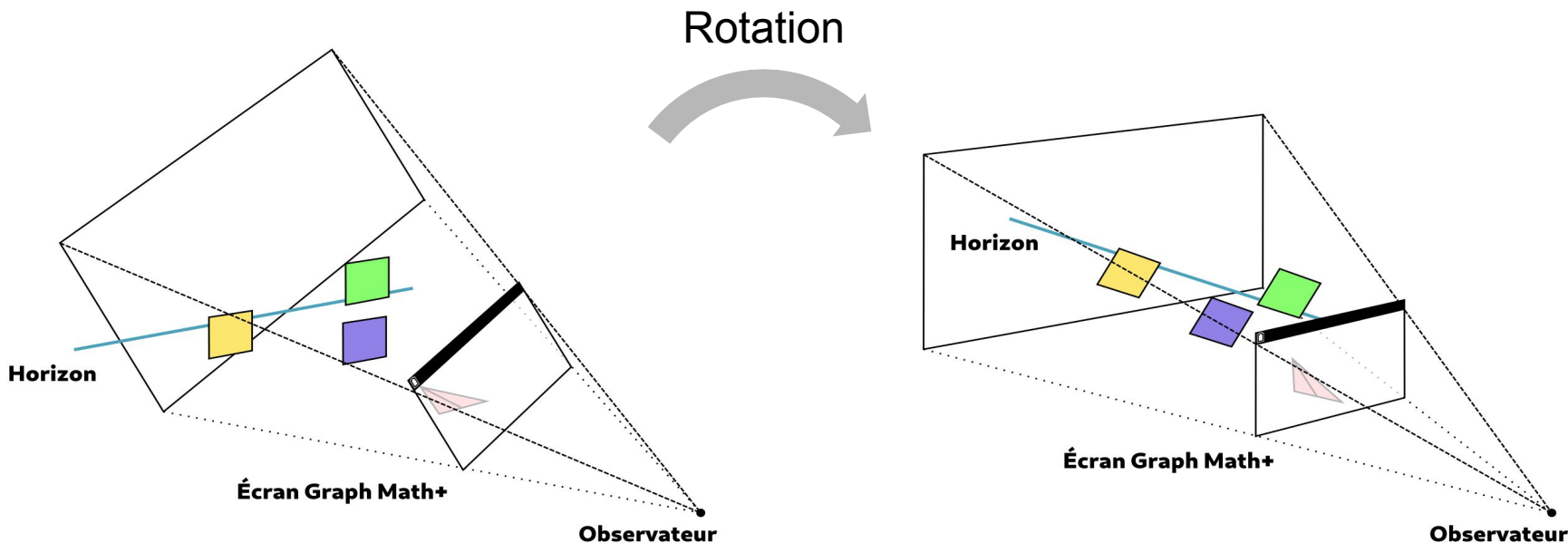
- Les objets du fond d'abord ⚠
- Ici, on triche : horizon en dernier (meilleure visibilité)



```
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  
89  
90  
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

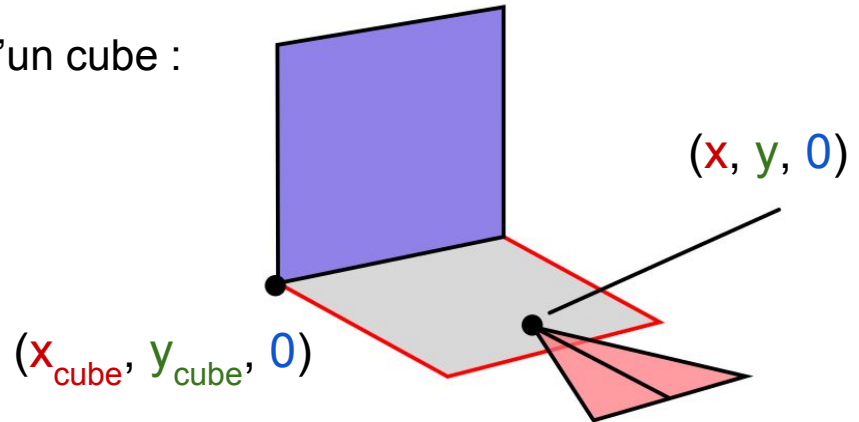
# Simulation physique

Déplacement des objets :

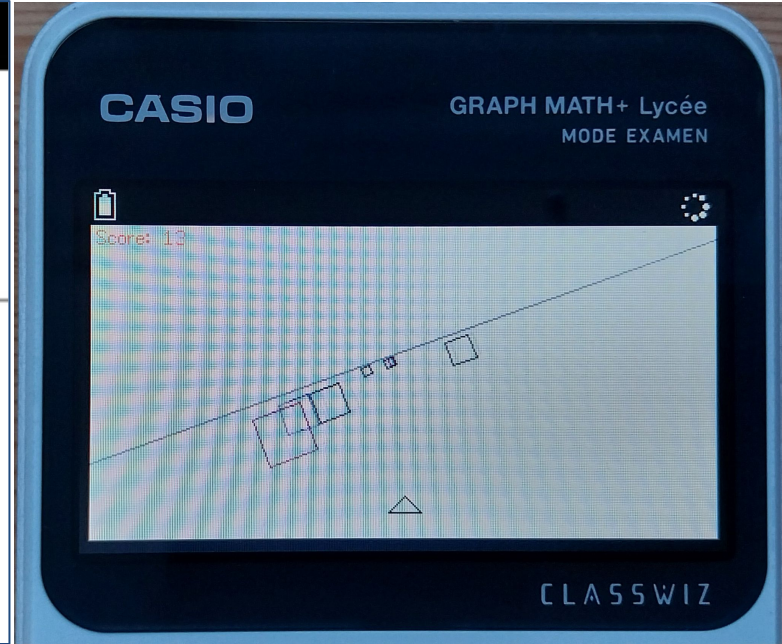
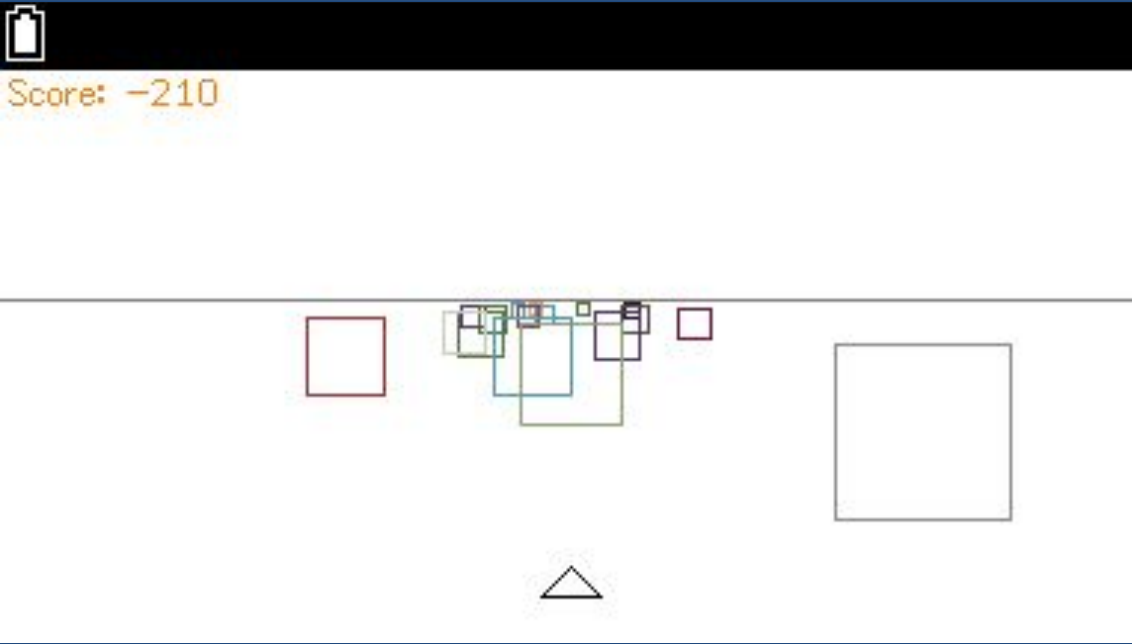
- On fait avancer l'observateur :  $y_{obs} += \langle \text{vitesse vaisseau} \rangle$
- Si on appuie à droite :  $x_{obs} += \langle \text{vitesse latérale} \rangle$   
... à gauche :  $x_{obs} -= \langle \text{vitesse latérale} \rangle$

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

- $x_{cube} \leq x \leq x_{cube} + \langle \text{largeur cube} \rangle$  et
- $y_{cube} - \langle \text{taille zone de danger} \rangle \leq y \leq y_{cube}$



# 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 :

Graph 35+E II	Graph 90+E	Graph Math+
<a href="#">Flappy Bird</a>	<a href="#">Flappy Bird</a>	<a href="#">Orlog</a>
<a href="#">1000 Bornes</a>	<a href="#">1000 Bornes</a>	<a href="#">WHIS</a>
<a href="#">Pykaster 3D</a>	<a href="#">Pykaster 3D</a>	<a href="#">Boules Bleues</a>
	<a href="#">Saute Mouton</a>	<a href="#">6 Qui Prend</a>
	<a href="#">Etiord City</a>	<a href="#">Jeu de Mémoire</a>
	<a href="#">Gravity Guy</a>	<a href="#">Jeu de mains...</a>
<a href="#">Synchro Donjon</a>		<a href="#">Couleurs RGB</a>
<a href="#">Alrys</a>		

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>