

DOCUMENTATIE : BUILDING THE GALAXY GAME

1. init_vertical_lines()

→ se creaza liniile verticale. Numarul liniilor verticale este dat de variabila `V_NB_LINES` [Number of Vertical LINES]. Ele vor fi retinute intr-o lista : `vertical_lines`.

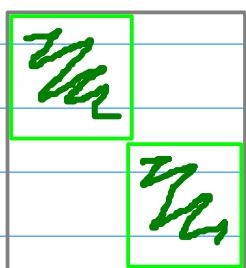
→ in kivy, fiecare widget are propriul canvas, adica o panza pe care se pot desena obiecte grafice. Prin sintaxa : `with self.canvas` noi spunem practic : vreau sa desenez pe canvasul acestui widget. Orice obiect grafic (Line - din kivy.graphics) instantiat aici va fi desenat in imaginea widgetului. Inainte de crearea obiectelor putem face mentiuni cu privire la aspectul lor. Ne putem gandi la o pensula si la mai multe tuburi de culori. Inainte sa desenez ceva, ma gandesc : ok, cu ce culoare sa desenez urmatorul obiect? Cam acelasi lucru se intampa si aici. Intai ii voi spune lui Kivy ce materiale sa foloseasca, iar el le va folosi cand noi vom instantia obiecte grafice.

2. transform(x,y)

→ functia de schimbare a perspectivei pentru un punct : coordonata lui x si coordonata lui y. Se folosesc doua functii auxiliare. Se returneaza rezultatul functiei auxiliare perspectivei dorite : `transform_2d` si `transform_perspective`. Schimbarea comportamentului functiei trebui facuta manual.

3. transform_2d(x,y)

→ converteste x si y la int, se rotunjeste astfel spre partea inferioara a numerelor : ce reprezinta pozitii de pixeli. Kivy accepta float, dar pixelii nu pot fi pe jumate ocupati (se va face anti-aliasing automat). Asadar, pentru o pozitie precisa de pixeli, convertim la int.



FLOAT

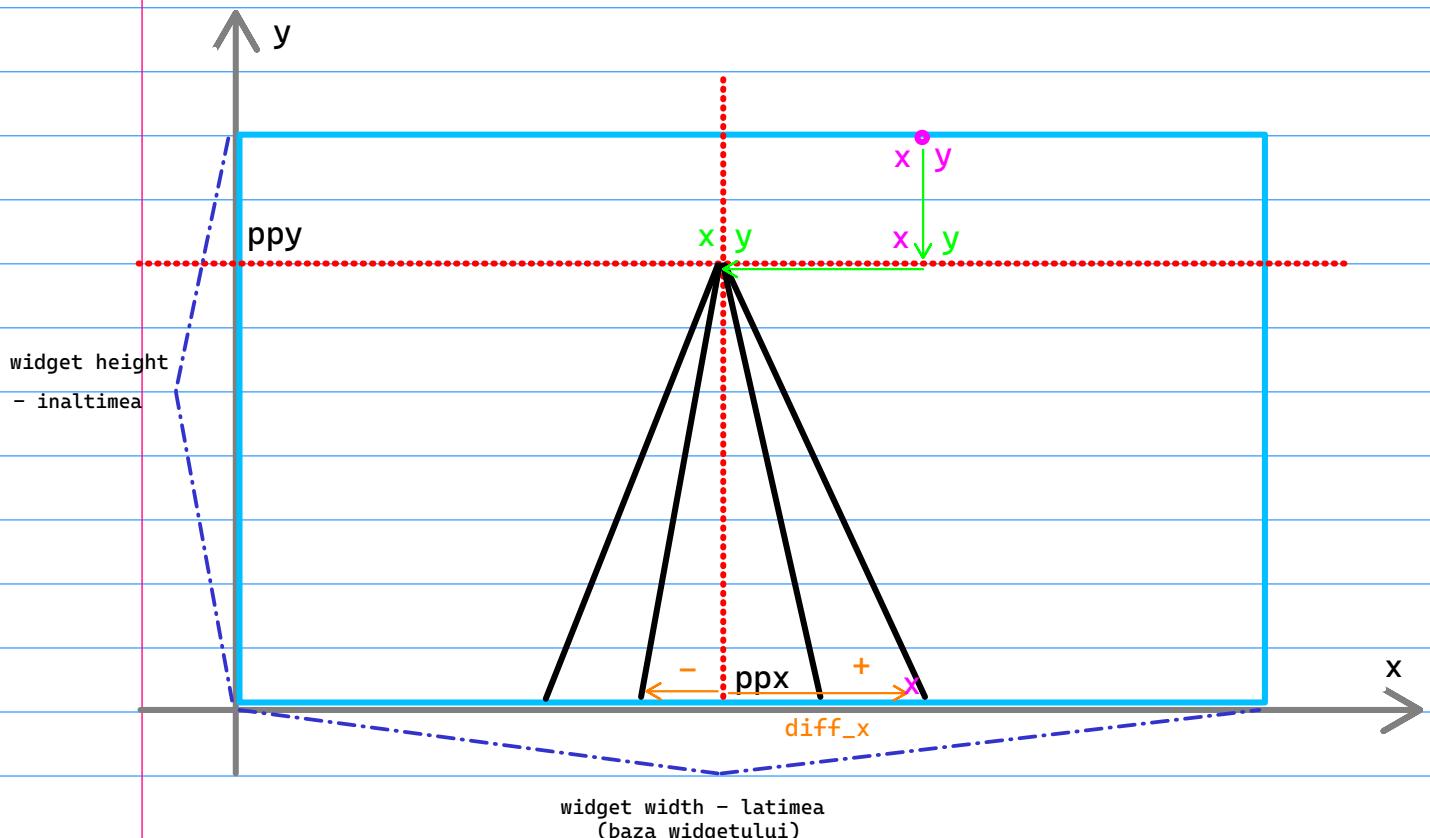


INT

4. `transform_perspective(x,y)`

→ primește un punct dat prin coordonate și recalculează poziția lui în perspectivă. Punctul de perspectivă este dat de variabilele : `perspective_point_x` și `perspective_point_y`.

→ `perspective_point_x` este linia de pixeli aflată la jumătatea latimii widget-ului (`width/2`), iar `perspective_point_y` este linia de pixeli aflată la 75% dinstanta de baza widgetului (`height*0.75`). Intersecția celor 2 cordoane este punctul relativ de perspectivă la dimensiunile widgetului.



Punctul de perspectivă : (ppx, ppy)

Punctul de intrare : (x, y)

`lin_y` - versiune scalată a lui y, mapată din intervalul [0, height] in intervalul [0, ppy]

$$\text{lin}_y = y * \text{ppy} / \text{height}$$

→ ppy este o pozitie de pixeli

ppy = height*0.75; EX : height = 1000 \Rightarrow ppy = pixeli de pe linia 750

Impartind ppy la inaltimea ecranului vom optine un procent (evident)

ppy/height = height*0.75/height = 0.75 (cat de aproape e punctul de perspectiva pe axa oy de inaltimea widgetului - valoare maxima)

→ inmultirea acestui procent cu y face ca aceasta sa isi modifice pozitia initiala cu 25% mai in jos pe axa oy.

→ astfel, ne asiguram ca orice y va fi pe ppy sau mai jos.

→ verificarea ca lin_y sa nu fie mai mare decat ppy e totusi necesara, vom vedea mai incolo de ce.

diff_x - cat de departe e punctul x de ppx : daca e cu plus inseamna ca suntem mai departe in dreapta, daca e cu minus suntem mai departe de el in stanga (in pixeli). $diff_x = x - ppx$

diff_y - distanta (in pixeli) de la ppy la lin_y. Pe desenul anterior aceasta distanta va fi 0. $diff_y = ppy - lin_y$

factor_y - procent : cat de departe e lin_y de ppy : $diff_y / ppx$

$diff = 0$, $ppy = 750$; $factor_y = 0$ (0%) \rightarrow pe desenul anterior

tr_x - coordonata x transformata. daca am aduna ppy cu diff_x am obtine aceiasi valoare input x. inmultind diff cu factor_y noi practic spunem : miscarea distanta dintre ppx si x pe masura ce lin_y e mai aproape de ppy. $tr_y = ppx + diff_x * factor_y$, pe desenul anterior $tr_y = ppx$

tr_y - coordonata y transformata. $tr_y = ppy - factor_y * ppy$
noi practic zicem : cu cat suntem mai aproape de ppy cu atat lin_y va ramane nemodificat. Totusi, acesta chiar va fi nemodificat in sensul ca : $factor_y = diff_y / ppy = (ppy - lin_y) / ppy$

$tr_y = ppy - factor_y * ppy$

$tr_y = ppy - [(ppy - lin_y) / ppy] * ppy$

$tr_y = ppy - ppy + lin_y = lin_y$

? Deci... ce am facut gresit? Nimic. Dar am sarit peste ridicarea la

putere a lui factor_y. De ce e acea linie de cod importantă? Vom vedea imediat.

5. init_horizontal_lines()

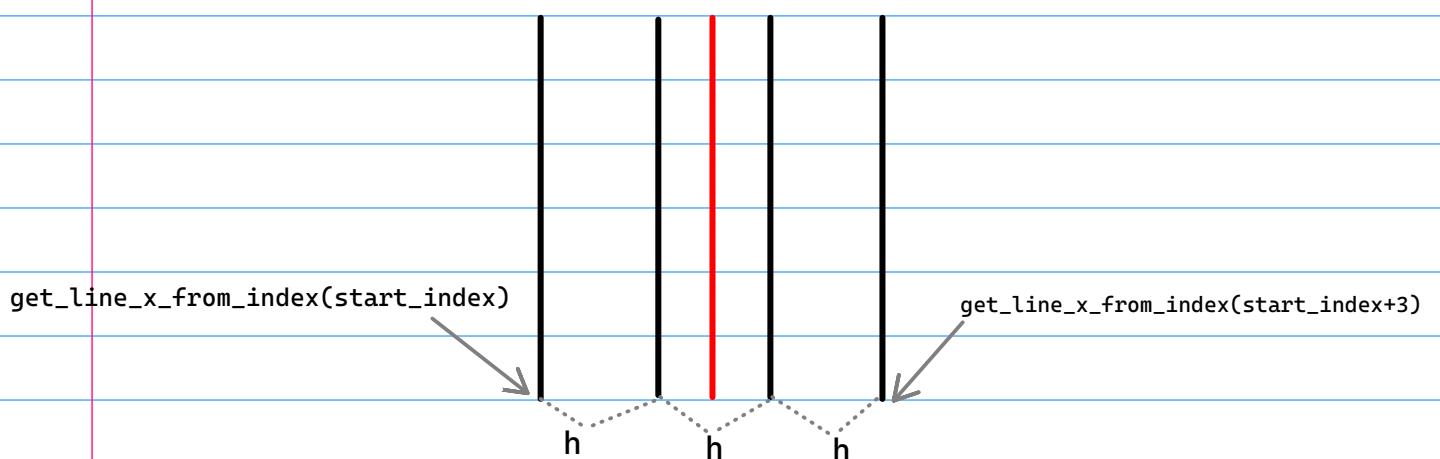
→ funcție pentru crearea liniilor orizontale. Numărul lor este dat de variabila `H_NB_LINES` (Number of Horizontal LINES), iar ele vor fi adăugate într-o listă : `horizontal_lines`

6. update_vertical_lines()

→ funcție care actualizează coordonatele liniilor verticale în scopul așezării lor corespunzătoare pe ecran. O linie este dată de 2 puncte, 4 coordonate : 2 coordonate pentru 'de unde începe linia', iar celelalte 2 'unde se termină linia'.

→ folosim 2 funcții auxiliare : transform - ne asigurăm ca punctele ce definesc liniile vor fi conform perspectivei dorite : 'de sus' sau 'spre orizont'; și `get_line_x_from_index`, care va returna poziția x a unei linii. Mai mult detalii despre această funcție imediat.

→ scop : noi vrem să pozitionăm liniile astfel :



→ linia roșie e ppx. vrem ca ppx să fie în mijlocul drumului din mijloc, și pe lângă drumul din mijloc să construim alte drumuri, la fel de largi ($h = h = h$). Pentru asta ne declarăm un `start_index`

→ Numărul de lini lini verticale e 4. Deci dacă le-am indexa, am avea :

lv1, lv2, lv3, lv4 = (1, 2, 3, 4). Noi vrem acum sa cream linia 1.

Asta face functia `get_line_x_from_index`. Returneaza X-ul liniei pe baza acestor indicii.

- se va stabili indexul de plecare, se vor parcurge toate liniile stocate din lista corespunzatoare si se va aplica transformarea pe perechile de puncte (`line_x, 0`) - linia incepe de la baza widgetului si (`line_x, height`) - linia se termina in tavanul widgetului.
- apoi vom seta coordonatele liniei la aceste valori.

7. `get_line_x_from_index(index)`

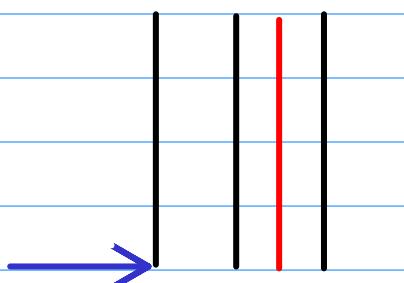
- returneaza coordonata x a unei linii verticale pe baza unor indice.

Noi vrem sa construim liniile simetric fata de ppx, iar pentru asta avem nevoie de o variabila `V_LINES_SPACING`, un procent care ne spune cat la suta din latimea widgetului vrem sa fie spatiul intre linii.

- atentie, numarul de linii verticale trebuie sa fie par, deoarece in caz contrar una dintre liniile nu va avea pereche de partea dreapta a lui ppx.

→ astfel, in prima faza calculam spatiul dintre liniile : `spacing = V_LINES_SPACING * width`

- inainte sa trecem mai departe, trebuie sa clarificam ceva



plecand de la indexul 1, cum stiu care va fi x liniei evidențiate?

e clar ca trebuie sa incep calculul cu ppx, deoarece in functie de el vreau sa construiesc. ppx - index * spacing - spacing/2? Ok, dar asta inseamna ca nu voi trece niciodata pe partea dreapta a lui ppx.

→ de aceea, formula de calcul a lui `start_index` arata : $-\text{int}(V_NB_LINES / 2) + 1$. Impartim numarul de linii verticale la 2. Asta ne va zice de unde incepem : -1, apoi 0 in partea stanga si 1, 2 partea dreapta.

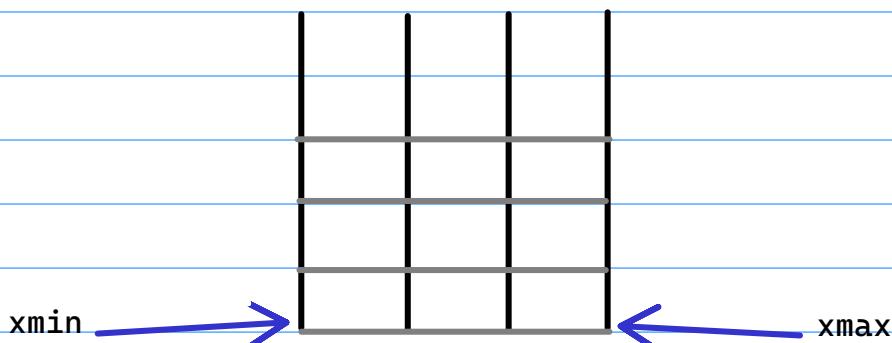
→ hai sa vedem : `ppx + index * spacing - spacing/2` arata mai bine, dar noi nu trebuie sa scadem $-\text{spacing}/2$ cand suntem in partea dreapta, ci sa adunam. De aceea apare variabila `offset = index - 0.5`. Practic noi spunem : scadem din index 0.5 ca sa includem acel `spacing/2`. Uite ce se intampla : $-1 - 0.5 = -1.5$; $-1.5 * spacing > e$ bine
 $0 - 0.5 = -0.5$; $-0.5 * spacing > e$ bine
 $1 - 0.5 = 0.5$; $0.5 * spacing > e$ bine, s.a.m.d

→ astfel am gasit formula pe care o returnam :

`line_x = ppx + (offset * spacing) + current_offset_x` (deocamdata nu ne intereseaza aceasta variabila)

8. `update_horizontal_lines()`

→ asemanator, functia aceasta actualizeaza datele despre pozitia liniilor pe widget. Deoarece vrem ca liniile orizontale sa taie liniile verticale, dar marginile lor sa nu depaseasca linia verticala cea mai din dreapta si cea mai din stanga, prima data v-am extrage coordonatele x ale acestor lini verticale.



`start_index` stim deja cum cum calculam, la `end_index` trebuie doar sa adunam `start_index` cu numarul de linii vericale si sa scadem 1. apoi apelam `get_line_x_from_index` cu aceste valori, pe rand, si vom optine

`xmin` (`maginea_stanag`) si `xmax` (`marginea_dreapta`).

→ la fel ca și la funcția de actualizare a liniilor verticale, avem nevoie de o funcție auxiliara `get_line_y_from_index` și clasica funcție `transform`.

→ se vor lua pe rand liniile stocate din lista corespunzătoare, vom lua coordonata `y` pentru fiecare linie cu `get_line_y_from_index`, vom transforma punctele `(xmin, line_y)` și `(xmax, line_y)` în perspectivă și vom seta noile coordonate liniei curente din interată `for`.

→ o mică precizare aici. intr-un array din py, dacă accesăm un element cu indice negativ vom optine : `[-1]` – ultimul element

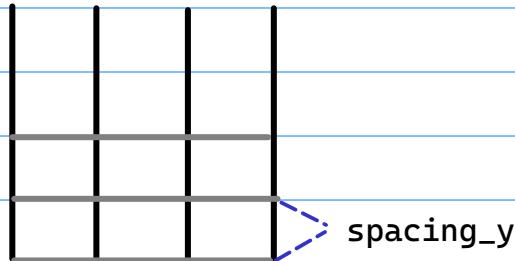
`[-2]` – penultimul, s.a.m.d

Deci în array-ul liniilor verticale, liniile din partea dreapta sunt stocate în partea stânga a array-ului, iar liniile din partea stânga sunt stocate în partea dreapta. Doar am vrut să precizez asta.

→ liniile orizontale vor fi stocate în ordinea lor care apar pe ecran, de sus în jos.

9. `get_line_y_from_index(index)`

→ observăm că liniile orizontale le indexăm de la 0 la `H_NB_LINES - 1`. (range-ul în py nu include și limita superioară).



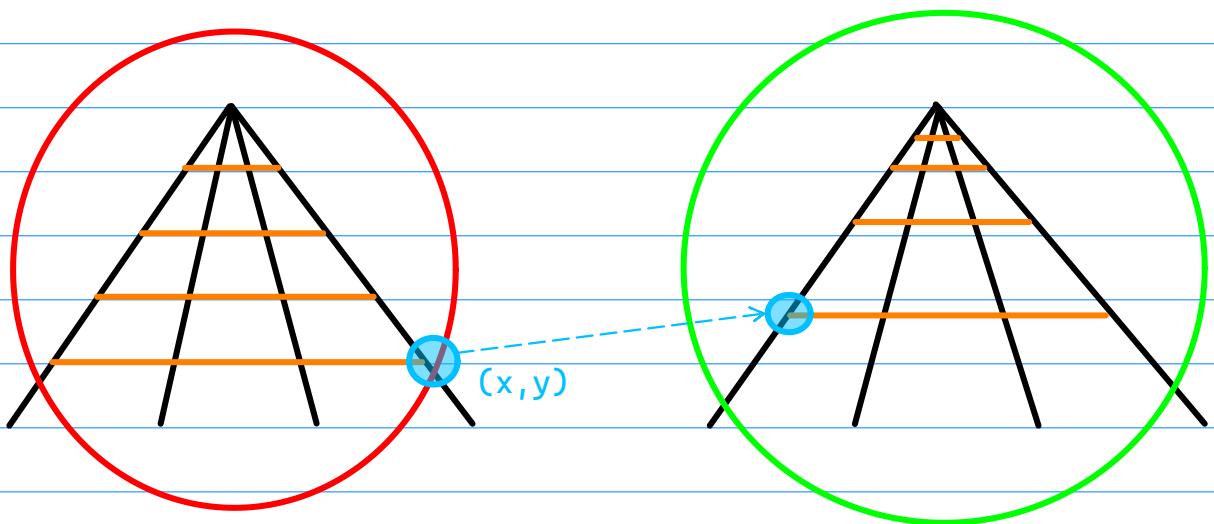
→ la fel, avem nevoie de o variabilă care să ne spuna că la sută din înaltimea widgetului va fi spațiul dintre liniile orizontale :

`H_LINES_SPACING`

→ astfel calculul nostru va fi destul de simplu : calculăm spațiul din-

liniile orizontale raportat la inaltimea ecranului : `spacing_y = H_LINES_SPACING * height;` apoi vom face pur si simplu : `line_y = index * spacing_y.` Despre variabila `current_offset_y` vom vorbi mai tarziu.

● In regula. Acum putem reveni la functia `transform_perspective`.
→ Ramasesem la idee ca, fara ridicarea la puterea 2 a lui `factor_y`, `tr_y = lin_y`. Ce ce am vrea sa ridicam `factor_y^2`?

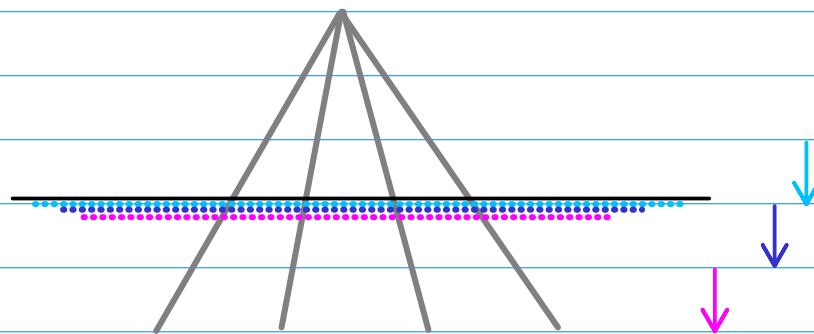


→ după cum putem observa în imaginea din stanga, distanța dintre liniile portocalii e constantă. Acest comportament nu reflectă pe indelete privirea din perspectiva, către orizont, deoarece în realitate placile care sunt mai aproape de noi se vad mai mari decât cele din departare.
→ noi deja stim ce înseamnă `factor_y`. Cât de departe e `lin_y (x,y)` de ppy. Sa presupunem ca în acest caz e 75% de departe față de ppy. Dacă inmultim $0.75 * 0.75 = 0.5625$. Asta înseamnă că ... punctul va fi mai apropiat de ppy. Exact ce ne dorim! Desenul poate nu ilustrează foarte bine aceste valori, însă acum funcția noastră de perspectivă arată mult mai bine!

10. `update(dt)`

→ pentru a simula mișcarea de deplasare în față a jucătorului, avem nevoie de o funcție care actualizează ecranul continuu la un interval fix

de timp. De ce? Deoarece noi trebuie modificam religios coordonate liniilor astfel incat ele sa se apropie din ce in ce mai mult de partea de jos a ecranului. Pentru a face asta, avem nevoie de o variabila `current_offset_y` : ne spune cu cat de mult fiecare linie s-a apropiat de originea ecranului.

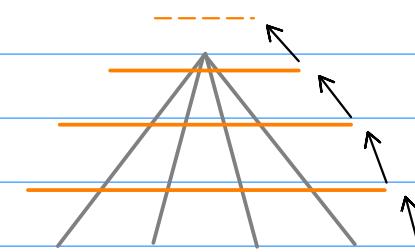


→ `Clock.schedule_interval(update, 1/60.0)` zice : functia update va fi apelata de 1/60.0 secunde. Asta inseamna 60 FPS. Iar noi in update vom modifica variabila `current_offset_y` corespunzator, dupa cum am vazut ca o folosim deja in functia `get_line_y_from_index`, unde calculam :

```
line_y = index * spacing_y - current_offset_y
```

(coordonata y a liniei va fi cu `current_offset_y` mai aproape de origine la fiecare apelare a functiei update, intrucat intuiim deja ca functiile de actualizare a liniilor vor fi si ele apelate in cadrul functiei update, la fiecare 0.016 secunde). La inceputul jocului, `current_offset_y = 0`.

→ Important e ce facem dupa ce prima linie a iesit din ecran. O varianta ar fi sa shift-am toate liniile mai sus cu o valoare constanta : spatiul dintre liniile orizontale.



→ Astfel putem simula miscarea in fata a liniilor. Un alt aspect de mentionat e ca avand cele 2 functii de actualizare a liniilor in functia update, la redimensionarea ecranului, toate obiectele vor fi aranjate proportional dupa noile dimensiuni ale acestuia. Super, nu?

→ In regula, acum hai sa discutam despre cum calculam efectiv acest offset. Dat fiind faptul ca cu cat offsetul e mai mare, viteza liniilor spre origine va creste, avem nevoie de o variabila `SPEED_Y` care va tine minte urmatorul procent : cat la suta din dimensiunea widgetului/ecranului va parcurge o linie intr-o secunda.

→ `dt` reprezinta timpul de la ultima apelare a functiei update. Valoarea pe care ne o asteptam ca acesta s-o aiba de fiecare data este 0.016.. Totusi, acest lucru nu se intampla. De ce? GPU-ul și sistemul de operare nu pot garanta o durată fixă pentru fiecare frame. Factori precum sarcinile de fundal, variațiile de încărcare grafică, alocările de memorie, throttle-ul termic sau prioritățile sistemului fac ca fiecare frame să dureze ușor diferit (ex.: 0.015–0.020 s). Asta inseamnă ca pe sistemele mai slabe, viteza jocului nostru va fi mai mică, dacă nu luăm și aceasta valoare în considerare.

→ `time_factor` = $dt * 60$. De ce inmultim cu 60? Vrem să vedem un procent care ne spune cat de lung sau cat de scurt a fost frame-ul anterior.

Ideal $dt = 1/60 \rightarrow time_factor = 1/60 * 60 = 1$ (100% : frame-ul a fost perfect). Dacă $dt > 1/60$; $time_factor > 1$ (frame-ul a fost mai lung)

→ In regula, acum hai sa calculam efectiv `current_offset_y`.

In `SPEED_Y` avem un procent, să zicem 0.4, care înseamnă 40% din înaltimea ecranului intr-o secundă. Deci $SPEED_Y * height$. Aceasta valoare trebuie să-o inmultim cu 100 deoarece noi avem nevoie de o valoare în pixeli cu care să mutăm liniile pe axa y la fiecare (aproximativ) 0.016 sec.

Ex : $0.4 * 800 = 320 / 100 = 3.2$ pixeli în 0.016 secunde

$3.2 * 60 * 0.016 = 320$ pixeli intr-o secundă

→ Totusi noi nu v-om avea norocul ca de fiecare data sa avem dt constanta, asa ca, ce trebuie sa facem e sa inmultim : $(SPEED_Y * height)/100$ cu time_factor. Astfel formula noastra de deplasare pe axa y va fi completa si corecta.

→ Ca sa simulam o scara infinita, vom verifica daca current_offset_y e mai mare decat spatiul dintre liniile orizontale, iar daca acest lucru se indeplineste v-om face current_offset_y - spacing_y. In caz contrar adaugam pixelii la current_offset_y.

→ Acum, hai sa ne concentrăm pe miscarea spre stanga sau spre dreapta. Intentia noastra este ca daca apasam pe partea stanga a ecranului sau pe sageata orientata spre stanga de pe tastatura, nava noastra, deocamdata imaginara, se va deplasa spre stanga, ramanand mereu centrata pe mijlocul ecranului.

→ Pentru a face asta avem nevoie de variabila : `current_offset_x`, dar si de alte catve functii.

11. `on_touch_up(touch)`

→ aceasta functie noi gestionam comportamentul miscarii cand degetul este ridicat de pe ecran : miscarea laterală se opreste : `current_speed_x` este 0.

12. `on_touch_down(touch)`

→ variabila touch contine toate datele despre momentul atingerii.

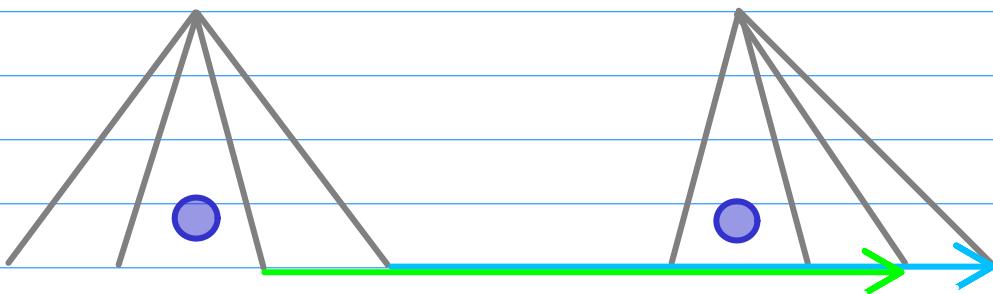
→ functia opusa lui `on_touch_up` evident ca va gestiona comportamentul miscarii cand degetul apasa ecranul.

→ verificam mai intai in ce parte a ecranului se efectueaza atingerea : daca coordonata x a atingerii se afla in partea stanga a ecranului atunci atribuim la `current_speed_x` valoarea pozitiva a lui `SPEED_X`, negativa in caz contrar.

→ `SPEED_X` reprezinta cati pixeli vom parcurge intr-un frame.

Consideram cazul ideal 1 frame = 0.016 s si `SPEED_X` = 1, conform formulei

din functia update vom face 900 (latimea ecranului) * 1 (SPEED_X) = 900 . Impatind la 100 va da 9 pixeli la fiecare frame. Asta inseamna 1% din latimea ecranului la fiecare 0.016 s. Daca SPEED_X este 1.4 asta va inseamna 1.4% din latimea ecranului la fiecare frame, nu 140% din latimea ecranului la fiecare secunda. Totusi, noi nu folosim direct SPEED_X in formula, ci current_speed_x, variabila care stie daca trebuie sa o luam la stanga (+), dreapta (-) sau sa stam pe loc (0). De ce facem asa?



Deoarece daca bila abastră se deplasează spre stanga, noi trebuie să creștem coordonata x a fiecarei linii, ca acestea să se repozitioneze în dreapta obiectului, după cum e ilustrat în desenul de mai sus.

```
line_x = center_line_x + (offset * spacing) + self.current_offset_x
```

→ linia : return super(RelativeLayout, self).on_touch_down(touch)
este esențială pentru a păstra comportamentul standard al widget-ului atunci cand suprascriem metoda on_touch_down. Fără ea touch-urile nu s-ar mai propaga în ierarhia de widget-uri, ceea ce ar impiedica alte componente să primească evenimentele de atingere, de exemplu butonul de start din ecranul de intampinare.

13. on_keyboard_up(keyboard, keycode)

→ exact același comportament ca on_touch_up; parametrii keyboard și keycode nu ne interesează

14. on_keyboard_down(keyboard, keycode, text, modifiers)

→ keycode este un tuplu furnizat de kivy cand se apasa o tasta.
(cod_tasta, nume_tasta). De aceea noi verificam keycode[1], sa vedem

daca s-a apasat ori tasta stanga, ori tasta dreapta, ajustand variabila current_speed_x corespunzator.

→ return True Spune Kivy că evenimentul a fost preluat și procesat, oprindu-i propagarea : doar miscarea spre stanga si dreapta vor fi afectate cand se vor apasa aceste taste. False raspandeste semnalul mai departe. Restul parametrilor din semnatura functiei nu ne intereseaza.

15. is_desktop()

→ verifica daca suntem pe un sistem de operare desktop : returneaza True. Returneaza False daca ne aflam pe un sistem de operare pentru dispozitivele mobile. Este necesar, deoarece in cazul in care utilizatorul ruleaza jocul de pe calculator, trebuie sa legam apasarea tastelor de functiile specifice.

→ Acest lucru se intampla in functia init prin liniile :

```
if self.is_desktop():
    self._keyboard = Window.request_keyboard(self.keyboard_closed, self)
    self._keyboard.bind(on_key_down=self.on_keyboard_down)
    self._keyboard.bind(on_key_up=self.on_keyboard_up)
```

→ la linia 2 cerem acces la tastatura; functia keyboard_closed va fi apelata cand aceasta este inchisa sau invalida, iar self este obiectul care primește evenimentele de la tastatura.

→ la linia 3 legam functia noastră on_keyboard_down de momentul cand se apasa o tasta

→ la linia 4 legam functia on_keyboard_up sa stim ce sa facem cand tastatura nu e apasata.

16. keyboard_closed()

→ dezlegam orice comportament de apasarea unei taste si definim comportamentul default pentru cand tastatura nu e apasata.

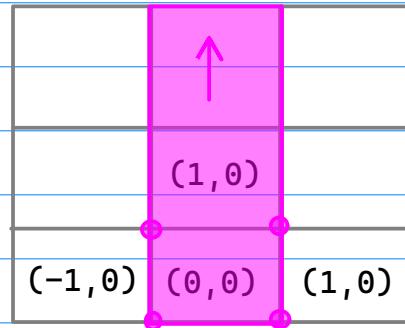
→ self._keyboard = None : nu mai e nevoie sa retinem obiectul care referentiază tastatura.

17. init_tiles()

→ functie care initializeaza placile albe ce vor forma drumul pe care nava noastra va trebui sa-l parcurga. Aceste vor fi de fapt niste trapeze, iar numarul lor va fi dat de variabila `NB_TILES`.

18. pre_fill_tiles_coordinates()

→ pentru afisarea placilor albe pe ecran vom avea nevoie de 2 array-uri. Unul in care tinem minte coordonatele placilor : `tiles_coordinates` si unul in care tinem minte obiectele : `tiles`. De ce? Pentru ca noi vrem sa tinem minte coordonatele trapezelor in acest format :



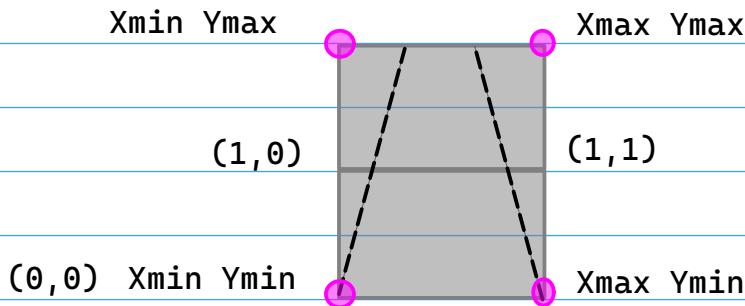
→ Coordonatele unui trapez reprezinta 4 puncte, 8 coordonate. Cand noi specificam locatia unui trapez pe ecran (unde va fi afisat) trebuie sa-i dam o lista de de genu : $[x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4]$.

→ Astfel, trebuie sa transformam coordonatele (X,Y) in 4 puncte valide.

→ functia `pre_fill_tiles_coordinates` este responsabila de a genera 10 coordonate pentru 10 tiles de asezat pe banda din mijloc (pentru ca inceputul sa nu ia jucatorul prin surprindere). Coordonatele de tipul (X,Y) pentru trapeze le vom numi coordonate de logică.

→ Folosim trapeze si nu dreptunghiuri, deoarece dreptunghiul e definit doar de 2 coordonate, insa noi cand vom trece in perspectiva 'spre orizont' vom aveam nevoie sa modificam toate cele 4 puncte ale figurii geometrice astfel obiectul sa apara potrivit pe ecran. Mai simplu : un dreptunghi privit din perspectiva 'spre orizont' pare un trapez.

19. update_tile()



→ functie care actualizeaza coordonatele placilor pentru afisarea lor corespunzatoare pe ecran. Cum?

→ interam prin array-ul de titles. Pe aceeasi pozitie i-a unui title se afla in array-ul de coordonate coordonatele de logica ale acestuia.

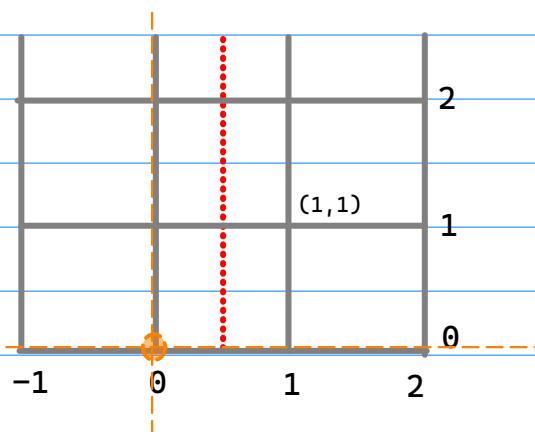
→ Avem nevoie de Xmin, Xmax, Ymin, Ymax, după cum ne dam seama de pe desen. Aceste vor fi aflate de funcția : get_tile_coordinates ce transforma coordonatele de logica în coordonate efective (de pixeli) pentru trapezele noastre.

→ Coordonatele de logica reprezinta de fapt colțul de stanga jos al figurii. Pentru a obține Xmax și Ymax trebuie să apelăm funcția get_tile_coordinates cu +1 pentru ambele coordonate.

→ Odată ce avem cele 4 coordonate putem scrie punctele ce definesc trapezul, transformându-le pentru afisarea din perspectivă.

20. get_tile_coordinates(ti_x, ti_y)

→ primim 2 coordonate de logica și întoarcem 2 coordonate ce reprezintă un punct pe ecran.



→ Noi deja avem functia `get_line_x_from_index` care primește un index de linie și întoarce coordonata x a acelei linii (pentru liniile orizontale) și functia `get_line_y_from_index` care întoarce coordonata y pentru un index primit (pentru liniile verticale).

→ Observăm că $(0,0)$ este punctul care are coordonata x a liniiei vericale indexată cu 0 și coordonata y a liniei orizontale indexate cu 0. La fel pentru punctul $(1,1)$.

→ Super, deci vom putea extrem de simplu să facem apel cu `ti_x` și `ti_y` la aceste funcții și vom primi coordonatele ce ne interesează.

21. `generate_titles_coordinates()`

→ funcție care generează drumul pe care trebuie să-l urmeze nava noastră

→ Un loop la noi înseamnă cand `current_offset_y ≥ spacing_y`. Cand un title ieșe din ecran, trebuie să-l stergem din lista noastră și să adăugăm altul la capatul superior (deasupra celorlalte), altfel am umplut memoria inutil. Deci, ne definim variabila `current_y_loop` care stie la care interacție ne aflăm. Initial, valoarea ei este 0. De fiecare dată cand o linie ajunge la baza ecranului, creștem numărul ei cu 1 și apelăm (în funcția `update`) funcția `data`, care se va ocupa de generarea infinită a drumului nostru astfel :

→ stergem ce nu ne trebuie, parcurgând lista de tiles invers. De ce? Dacă stergem un element dintr-o lista într-o loop `for`, lungimea ei se va modifica, iar operația repetitivă va sări peste elementul care a fost tras în stanga de acea stergere. Astfel, dacă coordonata y a placii este strict mai mică decât variabila `current_y_loop` – o stergem – (însemnă că a ieșit cu totul din ecran)

→ Observăm acum că, coordonatele brute y ale placilor, trebuie să fie din ce în ce mai mari. Dar noi trebuie să avem valori doar din intervalul $[0, V_LINES]$. Aceasta problema o rezolvăm simplu facând operația `ti_y = ti_y - current_y_loop` în funcția `get_tile_coordinates`;

→ Mai departe, scoatem coordonatele placii celei mai "din departare", si actualizam variabilele local last_x si last_y, unde last_y va fi incrementat cu 1.

→ Apoi trebuie sa ne decidem daca tile-ul pe care vrem sa il adaugam continua drumul tot inainte, face o curba la stanga sau face o curba la dreapta. Aceste directii vor fi generate prin valorile 0, 1 sau 2. Variabilele locale start_index si end_index asigura faptul ca nu vom genera urmatoarele placi in afara spatiului rezervat.

→ "last_x ≤ start_index" daca suntem lipiti de marginea stanga, vom face o curba catre dreapta garantat, iar daca last_x ≥ end_index vom face o curba la stanga garantat (nu vrem sa fim lipiti de perete, ci vrem sa directionam actiunea jocului pe mijloc)

→ o curba este formata din 3 placi (directie, fata, fata), de aceea avem acele 2 if-uri care verifica in ce parte vom contrui mai departe drumul nostru, in blocul carora mai adaugam inca 2 tile-uri pentru a construi corect curba.

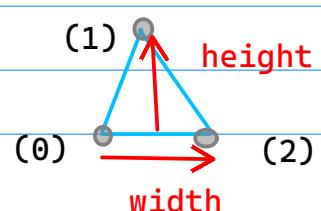
→ "last_y +=1" : observam ca ultima logica discutata e intr-un for. O curba candiese din ecran face sa dispara 2 placi din lista noastra, deci trebuie sa ne asiguram ca la fiecare generare, dupa stergere, vom adauga in vectorul de placi macar NB_TILES.

22. init_ship()

→ functie care deseneaza pe canvasul nostru un triunghi, de culoare alba. Tinem minte obiectul in variabila ship.

23. update_ship()

→ actualizeaza coordonatele navei noastre (chiar daca ea ramane tapana pe mijloc, avem nevoie de ea in primul rand ca sa ii definim pozitia pe ecran, si ca aceasta sa se actualizeze corect la redimensionarea ferestrei). Numarul de coordonate : 3



→ Ca să facem asta, avem nevoie de câteva variabile relative despre dimensiunile triunghiului, și una care să ne spuna cat de departe se află de baza ecranului. `SHIP_WIDTH` = 0.1; `SHIP_HEIGHT` = 0.035;
`SHIP_BASE_Y` = 0.04

→ Aflăm centru ecranului, aflăm cat de departe se află nava în raport baza ecranului, relativ la dimensiunea lui (4% din înaltime), apoi aflăm marimea ei, tot așa, relativ la dimensiunea ferestrei

→ Calculăm coordonatele efective pe ecran ale navei și actualizăm lista corespunzătoare : `ship_coordinates` , apoi dam aceste coordonate obiectului `ship`.

→ '*' despachetează tuplul

24. `check_ship_collision()`

→ verificăm dacă nava noastră nu a ieșit de pe drum (tot corpul trebuie să iasă de pe drum ca jucatorul să piardă)

→ "ti_y > current_y_loop + 1" asigură că verificăm coliziunile se face doar în interacția curentă. False înseamnă că e pe drum, True înseamnă că a ieșit de pe drum.

→ verificarea efectivă se face în funcția `check_ship_collision_with_tile`

25. `check_ship_collision_with_tile(ti_x, ti_y)`

→ extrag coordonatele colțului din stânga jos a placii și coordonatele colțului din dreapta sus a placii (coordonatele fizice)

→ Verific apoi toate coordonatele ship-ului, dacă se află pe acea placă.

Dacă da, returnez True. Contra, returnez False.

26. `reset_game()`

→ pur și simplu înnoieste toate datele pentru un joc nou

27. `clasa MenuWidget`

→ este widget-ul nostru care va gestiona contextul de intrare în joc, că și cel de ieșire (cand utilizatorul pierde). Acesta va sta întotdeauna peste ecranul propriu-zis al jocului, însă va fi vizibil doar în anumite

faze, prin atributul sau OPACITY (0 invizibil, 1 complet vizibil)

→ Faza A : Meniul de start. Contine un buton de start si afiseaza mare titlul jocului. Cand butonul este apasat, se executa functia `on_menu_button_press()`, care lanseaza sunete specifice in functie de valoarea buleana a variabilei `state_game_over`, care asa cum ii zice si numele, tine minte binar daca jocul a fost pierdut sau daca e pentru prima oara, dupa lansarea aplicatiei, cand utilizatorul apasa butonul de start. Se restarteaza jocul cu functia amintita mai devreme, si se schimba starea de adevar in variabila `state_game_has_started`, care stie daca jocul a inceput sau nu. In final, facem widgetul invizibil cu toti copii sai, iar datorita functiei `on_touch_down()` din clasa MenuWidget, semnalele de apasare a ecranului vor trece prin el si vor ajunge la widgetul central, aka MainWidget.

→ MenuWidget este copilul lui MainWidget. De aceea accesam, de exemplu, titlul jocului prin sintaxa `root.parent.menu_title` in fisierul menu.kv

→ Faza B : Meniul de restart. Se afiseaza acelasi buton, acelasi label, doar ca le sunt modificate textul. Inainte ca MenuWidget sa devina vizibil, in functia update, cand `check_ship_collision` returneaza false si `state_game_over` este false, schimbam proprietatile obiectelor noastre grafice, si anume textul, prin atribuirea lor cu un alt sir de caractere (GAME OVER pentru label si RESTART pentru buton). In final se lanseaza sunetele specifice, se opreste muzica de fundal si OPACITATEA lui MenuWidget devine iar 1.

→ functia `play_game_over_sound()` e intarziata cu 1 secunda, deoarece daca s-ar auzi imediat la oprirea jocului ar parea nesincronizat.

28. `init_audio()`

→ nimic special, doar incarca sunetele de care avem nevoie in variabile locale si seteaza volumul acestor (exprimat in procente)

29. `__init__(**kwargs)`

→ E ultima functie care ne-a mai ramas de discutat. Funcția `_init_` este constructorul clasei. Aceasta este prima funcție care se apelează automat atunci când se creează o instanță a jocului (porneste aplicatia). Rolul ei este să pregătească terenul: să initializeze variabilele, să configureze mediul grafic și să pornească motoarele jocului.

→ se apeleaza contructorul clasei parinte : `RelativeLayout`

→ se leaga metodele din celealte fisiere de clasa `MainWidget`, astfel pastram codul curat, fara a superpopula clasa

→ apelam toate functiile de initializare a obiectelor noastre vizuale

→ verificam pe ce dispozitiv ne aflam, deoarece pe un dispozitiv desktop vrem sa dam insemnataate tastelor sageti in ceea ce priveste miscarea navei

→ stabilim o fluiditate a jocului de 60FPS, programand ca functia `update()` sa se apeleze la fiecare 1/60 secunde

→ dam play la sunetul de 'bun venit' : `sound_galaxy`

30. Scorul

→ este un label scris in clasa `MainWidget`, care depinde de variabila `current_y_loop`. In `update`, ea se actualizeaza transforand numarul din aceasta in text pentru a scrie aceasta valoare in variabila `score_txt`, care este un `StringProperty`.

In regula, cred ca am discutat cam toate aspectele importante in ceea ce priveste prima versiune a aceste aplicatie, si care e logica ce o construieste. Multe multumiri fie aduse profesorului Jonathan Roux, cat si canalului de yt FreeCodeCamp pentru acest material educational.

GALAXY GAME V.1