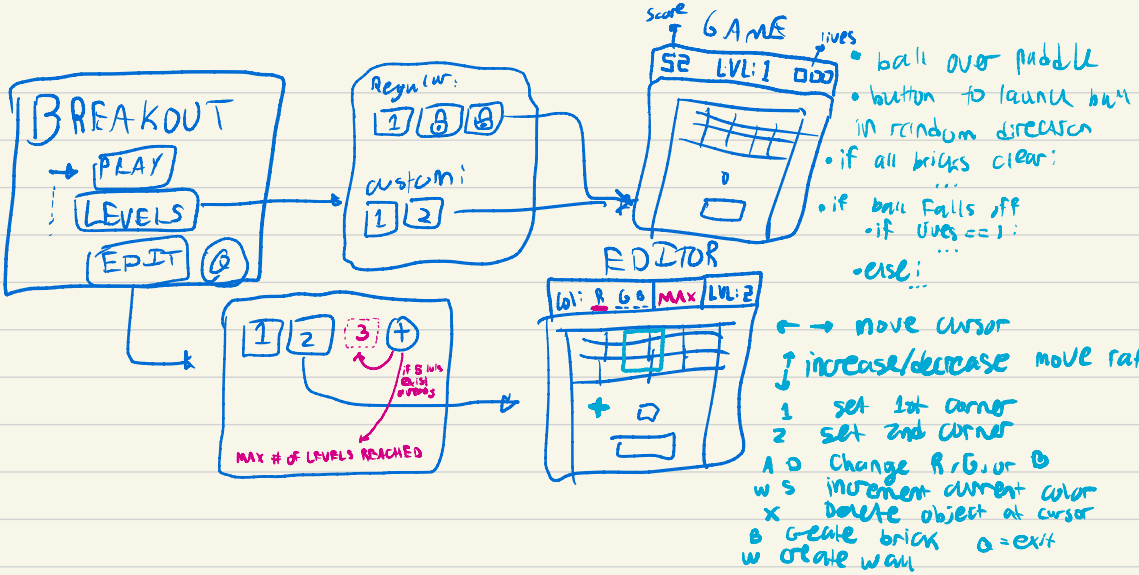
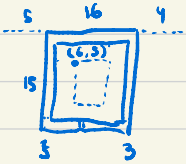


- ball over paddle
- button to launch ball in random direction
- if all bricks clear:
  ...
- if ball falls off
  - if lives == 1:
    ...
  - else:
    ...

← → move cursor
↑ ↓ increase/decrease move rate
1 set 1st corner
2 set 2nd corner
A D Change R, G, or B
W S increment current color
X delete object at cursor
B create brick Q = exit
W create wall

- Ball & paddle positions shown (cannot be changed)
- cannot overlap objects
- show rectangle between corner 1 & 2
- Autosaves level, (edits directly on array)
- Highlight selected color in RGB at top
- Display MAX temporarily if player tries to place wall or brick beyond array size



int = word POS = top left x, y : [0 → 127]

// No in between NULLs
Level regular_levels[5] (10620)
Level custom_levels[5] (10620)

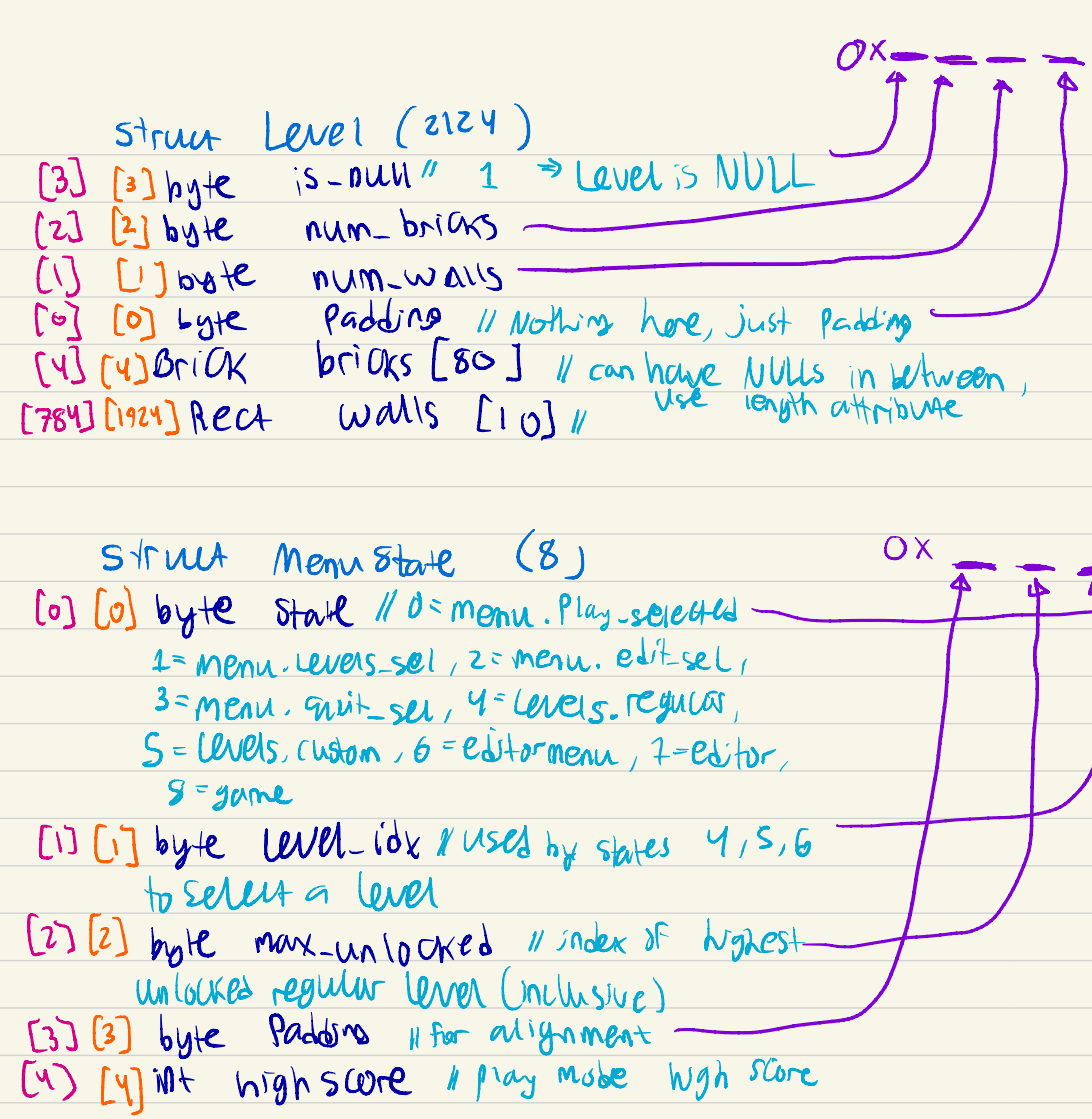
// Singletons
MenuState menuState
Game game
Edtor editor

```
struct Vec (8)
[0] [0] int x
[4] [4] int y

struct Rect (20)
          // ff0000 => Rect is NULL
[0] [0] int color
[4] [4] Vec Pos
[c] [12] Vec w-h

struct Brick (24)          // 0 = broken, -=1 on collision
[0] [0] int life   <- or null
[4] [4] Rect rect

struct bitmap
[0] [0] int width
[4] [4] int height
          // 0xff... means transparent
[8] [8] int pixels[]
```



GAME CONTROLS

- A D move Paddle
- A S W D menu navigation
- Space Action (select, launch ball, continue)
- R Go back (menu)

struct Game (2160)

[0] [0] level level

[84C] [2124] vec ball_pos init pos = (60, 102), dims = (3, 3)

[854] [2132] vec ball_vel

[85C] [2140] vec paddle_pos init pos = (50, 110), dims = (25, 3)

[864] [2148] byte mode_mask

least sig
- is_selected — // was level selected manually
- is_custom — // was the selected level custom

00 = play mode // 1st lvl → last
1 = selected regular level
2 = selected custom level
(back to menu after win)

[865] [2149] byte level_idx // stores current level being played, indexes into custom or regular based on mode_mask

[866] [2150] byte is_paused // 1 == paused

[867] [2151] byte padding // alignment

[868] [2152] int score // current score

[86C] [2156] int lives // if = 0, display Game over

0x

struct Editor // modifying in level array directly (32)

[0] [0] byte level_idx // idx into custom_levels array

[1] [1] byte RGB_sel // 0x01 = R, 0x02 = G, 0x03 = B

[2] [2] byte error_timer // decrements unless 0, if not 0, show "MAX" error up top

0x0n = lives

[3] [3] byte lives // alignment

[4] [4] vec cursor_pos

[C] [12] vec corner1 //

[14] [20] vec corner2

[1C] [28] int color

All allowed, account for this

0x

Δ state:

dirty = 1

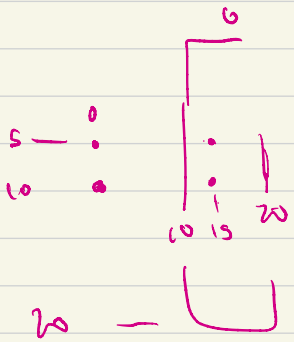
→ 8 (Game)
menu_state.state = 8
game.mode_mask = ⟨⟩
game.level_idx = ⟨⟩
game.score = ⟨⟩
game.lives = ⟨⟩

→ [0,3] (menu)
menu_state.state = [0,3]

→ [4,5] (level select)
menu_state.state = [4,5]

→ 6 (edit menu)
menu_state.state = 6

→ 7 (editor)
menu_state.state = 7
editor.level_idx = ⟨⟩



on dirty=1

dirty=0

clear_screen()

8 (Game)

game.level = copy(regular_levels[game.level_idx])
if game.mode_mask ∈ {00, 10}
else copy(custom_levels[game.level_idx])

game.ball_pos = (60, 102)

game.ball_vel = (0, 10)

game.paddle_pos = (50, 110)

draw_level(&game.level)

draw paddle, ball, lives, LVL, SCORE: <>

game.is_paused = 0

8 (editor)

editor.color = red

editor.RGB_sel = 1

editor.errorTimer = 0

editor.cursor_pos = (50, 50)

editor.corner1 = (10, 10)

editor.corner2 = (15, 15)

editor.dirs = 1

draw_editor()

[0, 3] (menu)

draw_menu() // menu, highscore, no outline

menu_update_outline()

menu_state.state = 0

[4, 5] (level select)

menu_state.state = 4

menu_state.level_idx = 0

draw_level_select() // all icons, no outlines

level_sel_update_outlines() // updates dynamic content

6 (edit menu)

menu_state.level_idx = 0

draw_edit_menu()

edit_menu_update_outlines()

fn_collide_level(level *, vec2 p, vec2 w_h)
→ void *, int

0 = NULL
No collision
Otherwise, ptr to collided obj in level

1 = brick
2 = wall
3 = possible/item
4 = no collision

// TODO: fix bitmap draw
Lock

fn_run_editor():
if dirty: follow procedure

if keyboard pressed:

if D & cursor.x < 127
    cursor.x++
elif A & cursor.x > 0
    cursor.x--
elif W & cursor.y > 0
    cursor.y--
elif S & cursor.y < 117
    cursor.y++

elif 1
    corner1 = cursor
elif 2
    corner2 = cursor

elif E & RGB_sel > 1
    RGB_sel--
elif F & RGB_sel < 4
    RGB_sel++

elif Q & ~~color[RGB_sel] < 255~~
    color[RGB_sel]++
elif E & ~~color[RGB_sel] > 0~~
    color[RGB_sel]--

elif R
    Return to editor menu

elif X
    for b in bricks
        if b.collide(cursor):
            b = NULL
            num_bricks--
            ; END
    for w in walls
        if w.collide(cursor):
            w = NULL
            num_walls--
            ; end

    END;

elif B
    if rect ! collide w/ walls & bricks
        find first NULL brick spot
        & write brick

    elif V
        same but for walls

return

fn_run_game()

if dirty:

execute instructions on last page

if !paused & !game over

compute ball unit vector $\vec{u}$

52,56 total_movement = (0,0)

while ~~total_movement < vel~~:

20,72 future_x = ball.x + $\vec{u}$.x

24,76 future_neg_x = ball.x + -$\vec{u}$.x

28,80 future_y = ball.y + $\vec{u}$.y

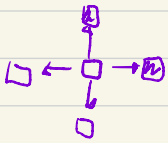
32,84 future_neg_y = ball.y + -$\vec{u}$.y

36,40 collide_x, collide_neg_x = 0

44,48 collide_y, collide_neg_y = 0

4 $\vec{u}$.x

8 $\vec{u}$.y



68 per

= total movement increment

60

for b in bricks:

- if collide(future_x): collide_x = 1
- if collide(future_neg_x): collide_neg_x = 1
- if collide(future_y): collide_y = 1
- if collide(future_neg_y): collide_neg_y = 1

① ⇒ b.life--, if b.life = 0:

draw_rect(erase, brick.rect), score++

update_score()

color -= $\frac{5}{6}$

64

for w in walls: ①

*show "complete" message

if paddle.collide(future_x): collide_x = 1

if paddle collide(future_neg_x): collide_neg_x = 1

if paddle collide(future_y): collide_y = 1

if paddle collide(future_neg_y): collide_neg_y = 1

if collide_x: ball.vel.x *= -1, $\vec{u}$.x *= -1

if !collide_neg_x: ball.pos.x += $\vec{u}$.x

else: ball.pos.x += $\vec{u}$.x

... same for y ...

dst
src
vel
trans
vel

draw_rect(erase, ball.pos, ball.w_h)
ball.pos += total movement
draw_rect(ball, ball.pos, ball.w_h)

erase & redraw score

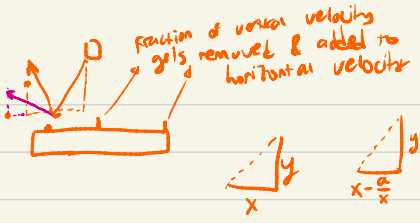
//state change
State = 0
dirty = 1

if keypress.a ^ paddle pos != max_left:
    erase paddle
    paddle pos --
    draw_paddle
same for right movement
if keypress_space & ball.vel = (0,0):
    ball.vel = random

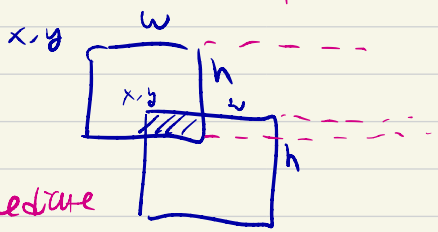
if ball out of bounds:
    game.lives --
    if lives > 0:
        // state change
        game.mode_next =
        game.level_idx = game.level_idx //no change
        game.score = game.score
        game.lives = game.lives
        dirty = 1
    else:
        draw_game_over()

---

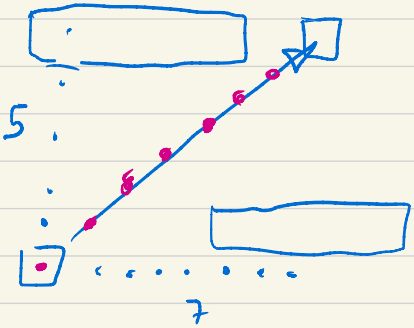
if keypress.p:
    toggle pause
if gameover & keypress.space:



② compute collision for each pt



① Generate intermediate pts



33

$-7, 3$

$\max(\text{abs}(x), \text{abs}(y)) = 7$

```
fn float_divide(num: int, divisor: int, accuracy: int) -> int
// return result of num ÷ divisor up to accuracy
// Return value must be shifted by accuracy to get actu
```

Var

return $num \cdot 2^{acc} \div divisor$

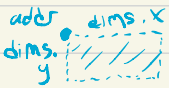
e.g $3 \div 7$, acc = 3

// not in code, mA uses mune

$\frac{3 \cdot 2^3}{7} = \frac{24}{7} = 3$

fn vec2addr (Vec) → int:
// convert vector to bit map address

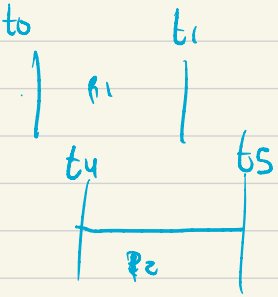
fn draw_int (Vec top_left, int val):
// using ~~addr as~~ top left display pos,
draw value in base 10
eg 0x00000 2a = 42 = !42!
div + mod useful for base conversion

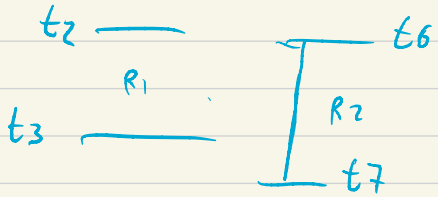
fn draw_rect (Rect rect):


draw rect

fn draw_bitmap (Vec top_left, int bitmap_addr):
// draws bitmap given using addr as top left
of screen

fn rect_collide (Vec r1_tl, Vec r1_wh, Vec r2_tl, Vec r2_wh) → int:
// return 1 if Rects collide, else 0





global scope call:

```
*$SP = P1
$SP -= 4
:
*$SP = Pn
$SP -= 4
jal fn(P1,...,Pn) → r1,...,rn
$SP += 4
save rn
:
$SP += 4
save r1
```

fn(P1,...,Pn) → r1,...,rn start

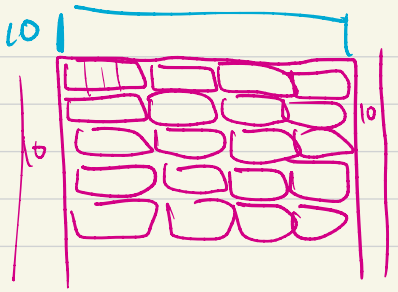
```
save Pn
$SP += 4
:
save P1
$SP += 4
```

fn(P1,...,Pn) → r1,...,rn return

```
*$SP = r1
$SP -= 4
:
*$SP = rn
$SP -= 4
j $ra
```

fn(P1,...,Pn) → r1,...,rn call from another fn

```
*$SP = $ra
$SP -= 4
*$SP = P1
$SP -= 4
:
*$SP = Pn
$SP -= 4
jal fn(P1,...,Pn) → r1,...,rn
$SP += 4
save rn
:
$SP += 4
save r1
$ra = $SP
$SP += 4
```



height = 118

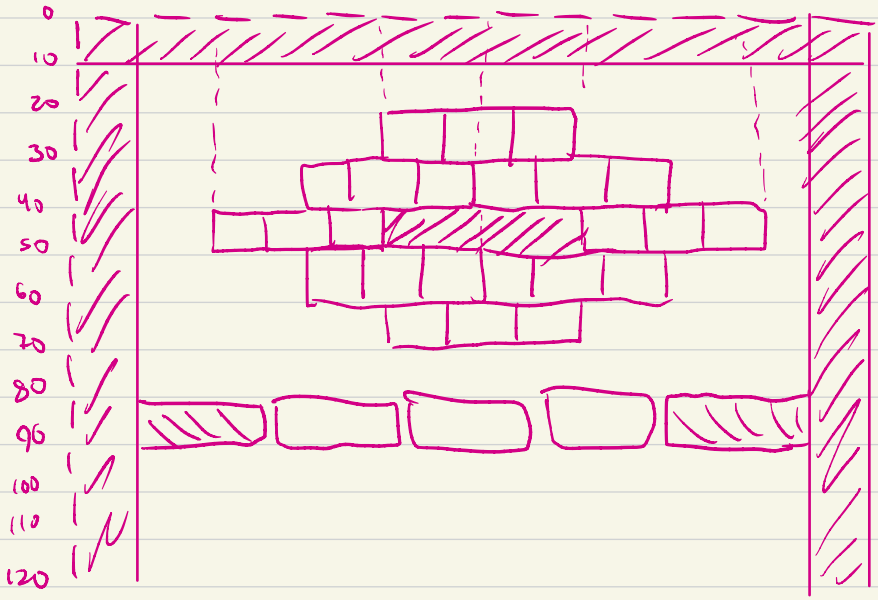
w = 108

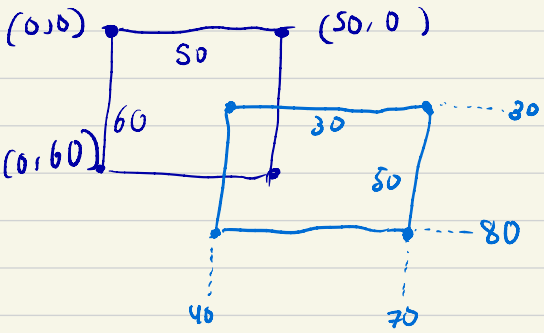
brick = (27, 10)

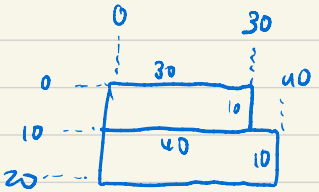
(0, 0), (10, 118)

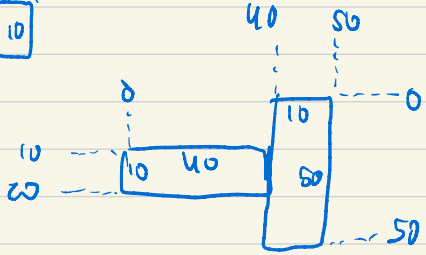
(118, 0), (10, 118)

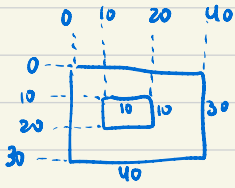
(10, 0), (108, 10)

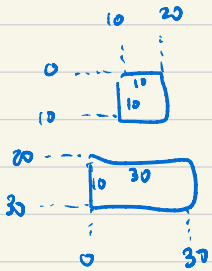



(0,0)
(50,0)
50
60
(0,60)
30
30
50
80
40
70


0
30
0
30
40
10
10
40
10
20


40
50
0
0
10
10
40
10
50
20
50


0
10
20
40
0
10
20
30
10
10
10
30
40


10
20
0
10
10
10
20
30
10
30
0
30

Interesting bug

sra will always round down, so $-0.99 = -1$
but $0.99 = 0$, so moving right will be slower!

```
                    ↱ arcade left shifts
fn round (value: int, shift: int)

    t0 = sra value, shift

    bound1 = sll t0, shift

    if bound1 == value:
        we know value is perfectly divisible, no rounding
        return t0

    bound2 = sll t0 + 1, shift

    diff1 = abs(value - bound1)
    diff2 = abs(value - bound2)

    if (diff1 < diff2):
        return t0
    else:
        return t0 + 1
```