Dinosaur Tactic

Detail Design Document

# I. Introduction:

An endless running game written in python with pyxel framework, features player as a running caveman trying to collect food while avoiding dinosaurs on his/her way.

# II. Gameplay:

* The player is chased by a hidden dinosaur (from the left side of the screen) and must run constantly
* Earning points by collecting food and running.
* Avoiding dinosaur coming up from the right screen side by navigating up and down. Also, player have a small booster system that can thrust them to the right.

Background pattern

Description automatically generated

# III. Game Logic

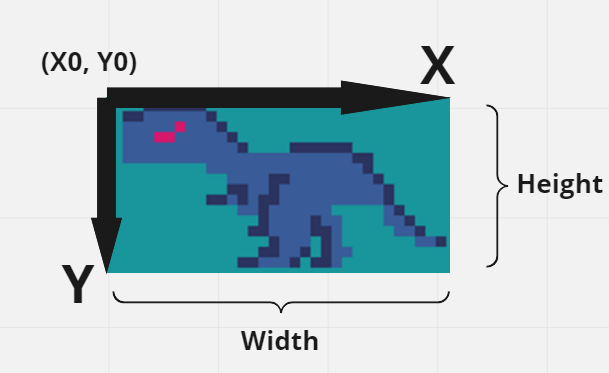
The pyxel framework provided a useful model to compile game with three underneath main functions: init, update, draw. To better utilized this design of the framework, it is better to design every entity in the game as an object that have three similar functions as well. Therefore, every entity in the game – player, dinosaur, and food all have the same layout of those function. However, each entity will can be added or removed properties that can be twerked to customize the entity’s behaviors (Will be discuss further).

Pyxel provide the class App(), which are the model discussed above, that act like an game engine that can render and update, responsible for setting up the program, receiving inputs while drawing and updating all entities at each frame.

Pyxel framework at its default will render 60 frame per second. This will be the timing system we can use by providing developer ability to get current frame counts since the program started.

*Coordinate:*

The canvas which the App() class draw upon has a coordinate system, which the origin at the top left corner. Each entity also has its own coordinate configuration, which combined with its height and width, to be drawn on the canvas.



*Animation:*

With the ability to get frame counts, animation can be created by drawing consecutive image of an entity that each image takes up a short period of time. The effect can be implemented by using mod calculation to set up different period for each image.

            if(pyxel.frame\_count % 20 > 7):

                pyxel.blt(self.x, self.y, 0, 16, 0, self.w, self.h)

            else:

                pyxel.blt(self.x, self.y, 0, 16, 8, self.w, self.h)

*Collision:*

The game’s only interaction between different entities is collision. This happens when the box border of an entity is overlapping with another. This can be implemented easily by checking entities’ coordinates, height and width. Example of a check between a dinosaur and the player’s box borders

self.player.x + self.player.w > dino.x

and dino.x + dino.w > self.player.x

and self.player.y + self.player.h > dino.y

and dino.y + dino.h > self.player.y

# IV. System design

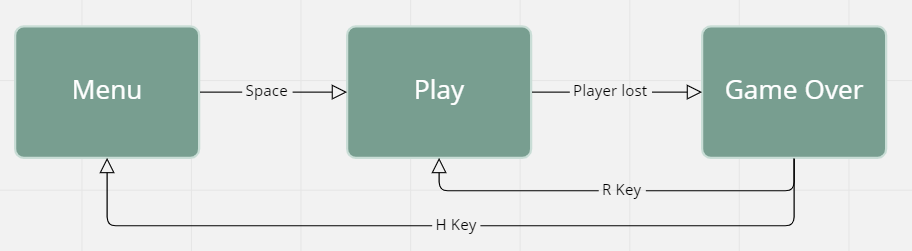
1. **Screen System & Game Flow**

The flow of the game can be represent with three difference scene: SCREEN\_MENU( a menu screen to display pre-game setting & high score), SCREEN\_PLAY (screen to play), and SCREEN\_GAMEOVER ( screen to display the game over state and present player with last game score vs high score). A property in the App() class called game\_state is used to control which screen is being render.

The enum class is created, called Game\_State, to better developing the game by using enum and using match case syntax.

As a result, the update and draw function of the App() class is also divided to three screen: update\_screen\_play(), update\_screen\_menu(), update\_screen\_over(), draw\_screen\_play(), draw\_screen\_menu(), draw\_screen\_over(). With this design, any more number of screen can be added in the future with ease if needed to (Ex: shop screen, inventory screen…)

The flow between screen can be presented by the diagram below and can be controlled by player input.



1. **Configuration & Utilities**

Screen Configuration Use:

SCREEN\_WIDTH = 240

SCREEN\_HEIGHT = 150

Game Configuration Use:

Player’s spawn coordinates:

PLAYER\_X\_START = 10

PLAYER\_Y\_START = SCREEN\_HEIGHT/2 - 8

Game’s speed and acceleration

GAME\_SPEED\_NORMAL = 0.5

GAME\_ACCELERATE\_NORMAL = 0.001

Global variables

high\_score = 0

current\_game\_speed = GAME\_SPEED\_NORMAL

current\_game\_acc = GAME\_ACCELERATE\_NORMAL

Since there are a lot of dinosaur and food entities that will be used in game, manipulating those entities by using arrays to store each kind of entity will produce much more optimize and clean code. Helper function is the key to implement this design. The first function is def update\_list(list) which will update every element inside the list. The next function is def draw\_list(list) will draw every element inside a list. The last one def cleanup\_list(list), which will remove all the element that is not being used by the game. This can be accomplished by marking every entities with is\_alive property, which will be set to False whenever that entity is not used. The function cleanup\_list(list) will then go in the array and remove all element that is mark as not alive (is\_alive = False). This help the game run faster and more optimize by removing all unused entities that can accumulate to be a large memory. Afterward, the array for each kind of entities can be set and ready to use.

A picture containing graphical user interface

Description automatically generated

There exist another helper function called reset\_game\_setting() that help resetting all global variables whenever the player start a new game.

1. **Music System**

All music sound set up can be found within the function sound\_set\_up(). In this function, two sfx effect sound (1st will play when player collected a food, 2nd will play when player die and blasts explode) and a music track is set up to be used in the framework. This function will be called with initialize the program. (in the init function of App() class)

The music sound can be turn on and off by player input in the menu screen (the information will be display as the sound icon and crossed sound icon), which achieved by setting a music property in App() class (True is on, False is off). The music control key is the M key. The music will be reset whenever the player started a new game while the sound effect will always be played in each game.

A picture containing text, first-aid kit, sign, clipart

Description automatically generated

1. **Endless Background**

The background itself is an entity that represented by the Background class. The background class have two coordinate data (x1, y1) and (x2, y2). This is used to create an endless running background effect by seamlessly moving 1st background toward the left side while the 2nd background append at the end of the 1st background (also moving to the left with same speed). When the 1st background moved all the way to the left and being hidden, it will be appended at the end of the 2nd background and this process will happen forever.

def update(self):

        self.x1 -= current\_game\_speed

        self.x2 -= current\_game\_speed

        if(self.x2 < 0):

            self.x1 = self.x2 + 256

        if(self.x1 < 0):

            self.x2 = self.x1 + 256

This process can be extent in vertical direction if the SCREEN\_HEIGHT is larger than the background image’s height by drawing two duplicate backgrounds that mimic the 1st and 2nd background behaviors underneath.

Diagram, schematic

Description automatically generated

1. **App Class**

The App class stand as a gateway to control which screen is being displayed on the canvas with help of the game\_state property. App class also store all variables and properties that need to ease the game experience: music, current score, last score….

*Property*

        self.background = Background()

        self.last\_score = 0

        self.music = True

  self.player = Player(PLAYER\_X\_START, PLAYER\_Y\_START)

        self.current\_score = 0

*Reset*

This function is used to reset all the game variable that need to start a new game (also call the reset game helper function to reset all global variable).

*Screens*

*Menu Screen*

Update: Listening to key input to change screen or mute music

Draw: Draw title, introduction, high score, background, and music control

*Game Screen*

Update: Update the game play, spawning in dinosaurs, foods, blasts, player, and background

Draw: All game entities and current score.

*Over Screen*

Update: Listening to replay or home control

Draw: Last score, high score, and guide text

1. **Points**

The game point is earned by two different ways:

* Player will earn point for every second they are playing
* Player will earn more bonus points the higher game speed they currently are playing at
* Player can earn point collecting food.

After losing, the new high score will be set if the current score is higher than current high score. High score is not stored within the program but only in the time the program is running.

if(int(pyxel.frame\_count / current\_game\_speed) % 60 == 0 or pyxel.frame\_count % 60 == 0):

            self.current\_score += 1

1. **Enemies System (Dinosaur)**

There are multiple dinosaurs that player need to avoid when playing. All of them will be spawning in the right side of the screen and then gradually move to the left side. If a dinosaur doesn’t collide with the player, it will be removed after moving across the screen. There are three types of dinosaurs with their own unique properties and behaviors. There is a limit to the maximum number of dinosaurs rendered in the game, which is 30. Every dinosaur spawned in the game will automatic append itself to the enemy array. All dinosaurs have animation(self) function to draw animation.

*Small Rex*

Spawn: Every 25 frames, on the right of the screen with random height

Property: Small dinosaur that move fast (8x8)

Behavior: Move toward the left side

*Big Rex*

Spawn: Every 80 frames, on the right of the screen with random height

Property: Bigger size, lower speed (16x32)

Behavior: Move toward the left side

*Flying Dinosaur*

Spawn: in packs, every 120 frames, on the right of the screen with random height

Property: Have acceleration, high speed

Behavior: Have a warning display for three second, then move toward the left side

A picture containing chart

Description automatically generated

1. **Food System**

Food is entity that player can collect to have more point. Each food collected can earn player 10 point and create a sfx sound. There is a maximum of number of foods that can be in the game, which is 30.

*Spawn*

A food entity will be spawned every 30 frames, which is about 0.5 second. A random integer will be generated to randomly choose a type for the food. The food will be spawn at a random height at the right side of the screen and will be moving to the left. Any food created will automatically be append to the food list. If the food is not collected after moving all the way to the left side of the screen, it will be removed.

r = random.randint(0, 3)

if(pyxel.frame\_count % 30 == 0 and len(food) < 30):

            new\_y = random.randint(0, SCREEN\_HEIGHT - 8)

            Food(SCREEN\_WIDTH - 8, new\_y, r)

*Type*

There are 4 different types of food, which can customize by sending input when creating the object as type = int from 0 to 3 (default is 0).

A picture containing text, first-aid kit, table, tiled

Description automatically generated

1. **Blasts**

The blast class represent the exploding entity, which created every time the player is collide with a dinosaur. The blast array is used instead of the a blast property in the App() class is due to the fact that player can be collide with multiple dinosaur upon death. This situation will spawn in a lot more blast to create dramatic effect when losing the game.

The blast will have a default configuration:

BLAST\_START\_RADIUS = 1

BLAST\_END\_RADIUS = 8

BLAST\_COLOR\_IN = 7

BLAST\_COLOR\_OUT = 10

As the blast spawned in, it will have a starting radius then expand to end radius and end its explosion. The game will switch to game over screen only when all the blasts’ explosion ended.

if(len(blasts) == 0 and not self.player.is\_alive):

            self.game\_state = Game\_State.SCREEN\_GAMEOVER

1. **Player**

The player is a special entity (the caveman) that can be controlled by player’s input.

*Spawn*

The player will be spawned in the middle of the SCREEN\_HEIGHT and in the left side of the screen. The spawning coordinate can by changed by changing configuration on PLAYER\_X\_START & PLAYER\_Y\_START. However, the player’s spawn is remaining the same for every game within a same configuration

*Animation*

The player’s caveman has two types of animation, which is the running animation and speeding animation. The speeding animation is only play when player hasn’t run out of booster yet and is pressing the D key. Two types of animation correspond to two animation function that share the same name of the types, which is call in the player’s draw function

*Control*

Player can control the caveman to move up and down with W and S key with no restriction expect for moving to the outside of the screen. Player also has a booster system that can thrust the player to move toward the right side in a short amount of time, which can be activated by Key D.

*Booster*

Player has a system that allows the player to move the caveman to the right side in a short amount. It is designed for the caveman to never exceed the 33.33% of the screen width. It takes roughly around 142 frames ~ 2.3 seconds for the player to refill their booster. If the player ran out of the booster, he can’t use it until some of the booster refilled. Also, if a player’s x-position is larger than player’s x-spawn-position, the player will be dragged back to their original position of PLAYER\_X\_START.

