Design Overview for Infinite Jumper

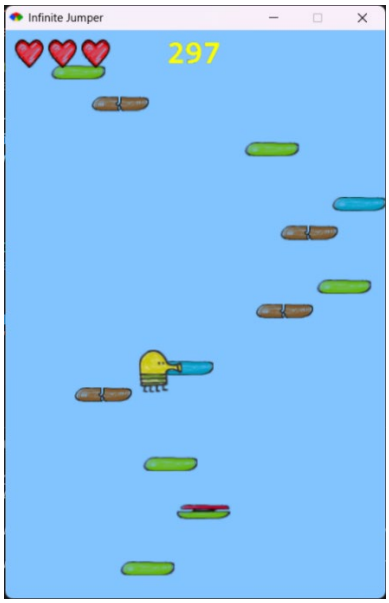
Name: Khang Vo
Student ID: 104220556

Summary of Program

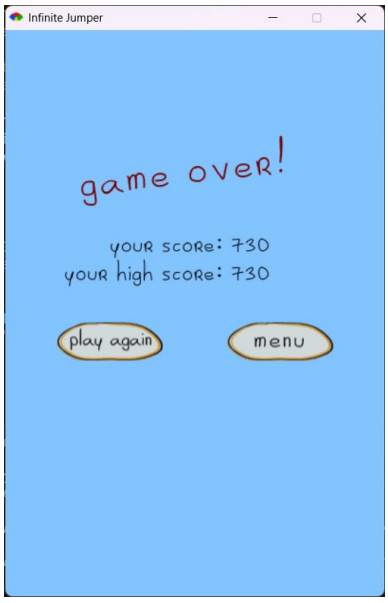
Infinite Jumper is a clone of the game “Doodle Jump”. In this game, user plays as a jumping character. The target of the user is to jump on floating platforms and jump as high as you can. There are two phases: the beginning phase with only normal platforms, and normal phase with all types of platforms. You can control your character to move left or right by using AD key or left, right arrow keys, and the character will automatically jump after landing on a platform. You have 3 hearts, and each time you touch a spike platform, you lose one. You die when you lose all 3 hearts, or when you fall out of the screen.

Screenshot:

1: Game menu



2: Game play (normal phase)



3: Game over

4: Static platform

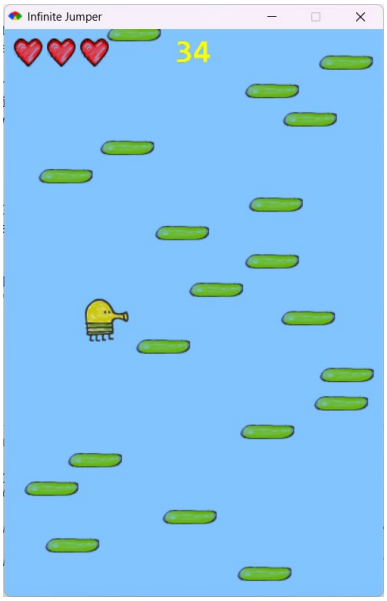
5: Spike platform

6: Boost platform

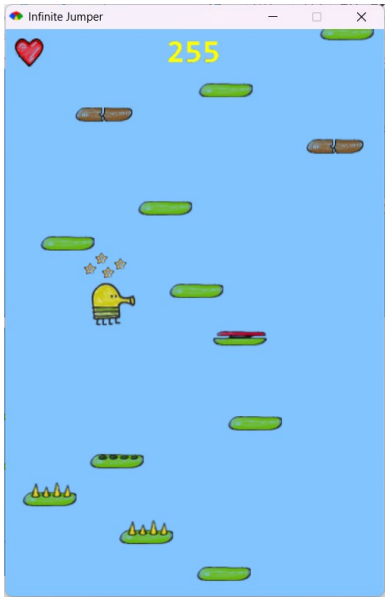
7: Horizontal movable platform

8: Vertical moveable platform

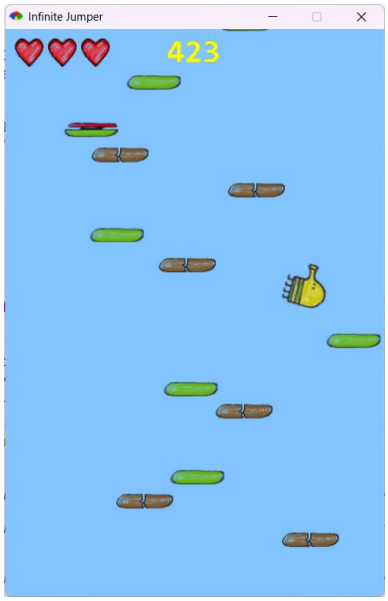
9: Breakable Platform



10: Beginning phase



11: Player is damaged



12: Player rolls while jumping

Required Data Types

In this game, I have a Player record to store all information that is related to the player character, such as position, size, score, health, etc.

Then, I have six different records for six different types of platforms. The reason why I use one record for each type of platform is because each type of platform will work in different ways, so they need unique variables to work. Also, creating different records for each type can make the code easier to understand.

Another record is the Button record, which is used to store information of buttons in game.

For enumeration, I only use Window, which contains the size of the window, and ZOrder, which contains different layers for the game. I used to use an enumeration to store platform's types, but I changed to use Symbol for simplicity and efficiency.

Table 1: Player
This record stores images, position, size, velocity, hitbox, score, and health of the player

Field Name	Type	Notes
score	Integer	The current score of player
img_left	Gosu::Image	Player's sprite facing left
img_right	Gosu::Image	Player's sprite facing right
img_stars	Gosu::Image	Player's dizzy stars sprite
img_heart	Gosu::Image	Player's heart sprite
sfx_jump	Gosu::Sample	Player's jumping sound
x	Float	x ordinate of player
y	Float	y ordinate of player
w	Integer	Width of player

h	Integer	Height of player
vx	Float	Horizontal velocity
vy	Float	Vertical velocity
ax	Float	Horizontal acceleration
ay	Float	Vertical acceleration
heart	Integer	Player's hearts
score	Integer	Player's score
font_score	Gosu::Font	Score font
dir	String	Player's facing direction
roll	Integer	Time when player rolls
time_start_hurt	Integer	Time when player is hurt
top	Float	Player's top ordinate
bottom	Float	Player's bottom ordinate
left	Float	Player's left ordinate
right	Float	Player's right ordinate

Table 2: Static Platform
This record contains platform's type, image, position, size, and hitbox

Field Name	Type	Notes
type	Symbol	Type of platform
img	Gosu::Image	Platform's sprite
x	Float	x ordinate of platform
y	Float	y ordinate of platform
w	Integer	Width of platform
h	Integer	Height of platform
top	Float	Platform's top ordinate
bottom	Float	Platform's bottom ordinate
left	Float	Platform's left ordinate
right	Float	Platform's right ordinate

Table 3: Spike Platform
This record contains platform's type, image, position, size, hitbox, sound effect, a spike variable and timing variable.

Field Name	Type	Notes
type	Symbol	Type of platform
img_normal	Gosu::Image	Normal platform's sprite
img_active	Gosu::Image	Active platform's sprite
img	Gosu::Image	Platform's current sprite
sfx	Gosu::Sample	Platform's sound effect
x	Float	x ordinate of platform
y	Float	y ordinate of platform
w	Integer	Width of platform
h	Integer	Height of platform
is_spike	Boolean	The platform is active or not
start_delay	Integer	Time when player land on

delay_time	Integer	Delay before change state
top	Float	Platform's top ordinate
bottom	Float	Platform's bottom ordinate
left	Float	Platform's left ordinate
right	Float	Platform's right ordinate

Table 4: Boost Platform
This record contains platform's type, image, position, size, hitbox, and sound effect

Field Name	Type	Notes
type	Symbol	Type of platform
img_active	Gosu::Image	Active platform's sprite
img	Gosu::Image	Platform's normal sprite
sfx	Gosu::Sample	Platform's sound effect
x	Float	x ordinate of platform
y	Float	y ordinate of platform
w	Integer	Width of platform
h	Integer	Height of platform
top	Float	Platform's top ordinate
bottom	Float	Platform's bottom ordinate
left	Float	Platform's left ordinate
right	Float	Platform's right ordinate

Table 5: Horizontal Moveable Platform
This record contains platform's type, image, position, size, hitbox, direction and velocity.

Field Name	Type	Notes
type	Symbol	Type of platform
img	Gosu::Image	Platform's sprite
x	Float	x ordinate of platform
y	Float	y ordinate of platform
w	Integer	Width of platform
h	Integer	Height of platform
dir	Integer	Platform moving direction
vx	Integer	Platform horizontal velocity
top	Float	Platform's top ordinate
bottom	Float	Platform's bottom ordinate
left	Float	Platform's left ordinate
right	Float	Platform's right ordinate

Table 6: Vertical Moveable Platform
This record contains platform's type, image, position, size, hitbox, direction, velocity, and a timing variable

Field Name	Type	Notes
type	Symbol	Type of platform

img	Gosu::Image	Platform's sprite
x	Float	x ordinate of platform
y	Float	y ordinate of platform
w	Integer	Width of platform
h	Integer	Height of platform
dir	Integer	Platform moving direction
vy	Integer	Platform vertical velocity
t	Integer	Time since last change direction
top	Float	Platform's top ordinate
bottom	Float	Platform's bottom ordinate
left	Float	Platform's left ordinate
right	Float	Platform's right ordinate

Table 7: Breakable Platform
This record contains platform's type, images, position, size, hitbox, sound effect, a broken variable and dropping velocity

Field Name	Type	Notes
type	Symbol	Type of platform
imgs	Array[Gosu::Image]	Platform's sprites
img	Gosu::Image	Platform's current sprite
sfx	Gosu::Sample	Platform's sound effect
x	Float	x ordinate of platform
y	Float	y ordinate of platform
w	Integer	Width of platform
h	Integer	Height of platform
broken	Integer / nil	Time since break platform
vy	Integer	Platform vertical velocity
top	Float	Platform's top ordinate
bottom	Float	Platform's bottom ordinate
left	Float	Platform's left ordinate
right	Float	Platform's right ordinate

Table 8: Button
This record contains button's image, position, and size

Field Name	Type	Notes
x	Float	x ordinate of button
y	Float	y ordinate of button
w	Integer	Width of button
h	Integer	Height of button
normal	Gosu::Image	Normal button's image
pressed	Gosu::Image	Pressed button's image
img	Gosu::Image	Button's current image

Table 9: Window details
This enumeration contains window size

Value	Notes
WIDTH	Window's width
HEIGHT	Window's height

Table 10: ZOrder details
This enumeration contains different layers of the game

Value	Notes
BACKGROUND	Background layer
PLATFORMS	Platforms layer
PLAYER	Player layer
UI	Interface layer
OVERLAY	Top overlay layer

Overview of Program Structure

- This game has three separate states: when open the game, it will enter **Menu State**, then pressing play button will make it switch to **Play State**, and the final is the **Replay State** to show score after a game.
- In Play State, an **initialize** method is used to generate the list of platforms and the player. The **draw** method will draw all the platforms and the player onto the screen, and the **update** method will animate and handle physics and collisions between player and platforms.

Main functions/procedures:

[State-related functions]
1. Main window with state

```ruby
# Main game window
class MainWindow < Gosu::Window
  attr_accessor :state

  def initialize
    super Window::WIDTH, Window::HEIGHT
    self.caption = "Infinite Jumper"
  end

  def draw
    @state.draw
  end

  def update
    @state.update
  end
end
```

2. Leave the old state and switch the window's state to the new_state

state_switch (window, new_state)

```ruby
# Switch the current state of window
# @param window: the window to change state
# @param new_state: the state to change to
def state_switch(window, new_state)
  window.state && window.state.leave
  window.state = new_state
  new_state.enter
end
```

[Player's functions]

1. Make the player to roll

roll (player)

degree_since_roll (player) **-> integer**

```ruby
# Roll the player
# -> Set the time when player starts to roll to calculate the player angle
later
# @param player: the player to roll
def roll(player)
  player.roll = Gosu.milliseconds
end

# Calculate the player's angle since when the player starts to roll
# @param player: the player to calculate the angle
def degree_since_roll(player)
  time_passed = Gosu.milliseconds - player.roll
  if time_passed > 740
    player.roll = nil
    return 0
  else
    return -360 * (1.0 - (1.0 - time_passed.to_f / 740.0) ** 2.0)
  end
end
```

2. Damage the player, reduce player's heart by 1

damage (player)

```ruby
# Damage a player
# -> reduce player's heart by 1
# @param player: the player to damage
def damage(player)
  player.heart -= 1
  player.time_start_hurt = Gosu.milliseconds
end
```

3. Animate the player to fall under gravity

fall (player)

```ruby
# Make a player to fall
# -> add gravity to player's vertical velocity
# @param player: the player to fall
def fall(player)
  player.vy += player.ay
end
```

4. Make player to jump with initial velocity vy, and set the volume of jumping sound effect to vol

jump (player, vy, vol)

```
# Make a player to jump
# @param player: the player to jump
# @param vy: the jumping velocity (negative value). The bigger -vy is, the
higher the player jump. Default to -11
# @param vol: the volume of the jumping sound effect. Set to 0 to mute sound
effect. Default to 1
def jump(player, vy = - 11, vol = 1)
  player.vy = vy
  player.sfx_jump.play(vol)
end
```

5. Move the player with small acceleration, change the player's facing direction and update the player's horizontal velocity

move_left (player)

move_right (player)

```
# Move a player left, also change the player's direction
# @param player: the player to move
def move_left(player)
  player.dir = 'left'
  if player.vx > -6
    player.ax = -0.4
    player.vx += player.ax
  end
end

# Move a player right, also change the player's direction
# @param player: the player to move
def move_right(player)
  player.dir = 'right'
  if player.vx < 6
    player.ax = 0.4
    player.vx += player.ax
  end
end
```

6. Update the player position and hitbox according to the updated velocity

player_move_x (player)

player_move_y (player)

```
# Horizontally move a player
# @param player: the player to move
def player_move_x(player)
  player.x += player.vx

  player.x = player.x % Window::WIDTH

  player.left = player.x - 15
  player.right = player.x + 15
end

# Vertically move a player
# @param player: the player to move
def player_move_y(player)
  player.y += player.vy

  player.top = player.y - player.h/2
```

```
  player.bottom = player.y + player.h/2
end
```

7. Check if player is colliding with platform using hitboxes

`collide_with` (player, platform) **-> boolean**

```
# Check if a player is colliding with a platform
# @param player: the player to check
# @param platform: the platform to check
def collide_with(player, platform)
  if (platform.left <= player.left and player.left <= platform.right) or
(platform.left <= player.right and player.right <= platform.right)
    if platform.bottom >= player.bottom and player.bottom >= platform.top
      return true
    end
  end
  return false
end
```

8. Draw the player on the screen

`player_draw` (player)

```
# Draw a player on the screen
# @param player: the player to draw
def player_draw(player)
  case player.dir
  when 'left'
    img = player.img_left
  when 'right'
    img = player.img_right
  end
  if is_hurt(player) and ((Gosu.milliseconds - player.time_start_hurt) / 50)
.to_i.even?
    opacity = 0x66_ffffff
  else
    opacity = 0xff_ffffff
  end
  if player.roll == nil
    img.draw_rot(player.x, player.y, ZOrder::PLAYER, 0, 0.5, 0.5, 1, 1,
opacity)
  else
    img.draw_rot(player.x, player.y, ZOrder::PLAYER, degree_since_roll(player),
0.5, 0.5, 1, 1, opacity)
  end
  player.img_stars.draw_rot(player.x, player.top, ZOrder::PLAYER, 0, 0.5, 0.5,
1, 1, opacity) if is_hurt(player) or is_dead(player)
end
```

9. Draw the score at the top of the screen

`draw_score` (player)

```
# Draw a player's score on screen
# @param player: the player whose score is drawed
def draw_score(player)
  player.font_score.draw_text(player.score.to_s, 200 -
player.score.to_s.length*10, 10, ZOrder::UI, 1, 1, Gosu::Color::YELLOW)
end
```

10. Draw player's heart at the top left of the screen

`draw_heart` (player)

```
# Draw a player's hearts on screen
# @param player: the player whose hearts are drawed
```

```ruby
def draw_heart(player)
  player.heart.times do |i|
    player.img_heart.draw(10 + i*35, 10, ZOrder::UI)
  end
end
```

[Platform's functions]

1. Generate a random set of platforms of different types at random position above the screen within the limit. `last_x` is the x ordinate of the current highest platform.

`generate_random_standable_platform (last_x, limit)-> Array[Platform]`

```ruby
# Generate random standable platforms, including spike, boost, moveable and
normal platforms
# @param last_x: The x ordinate of the highest platform
# @param limit: The horizontal distance limit between old and new platforms
def generate_random_standable_platform(last_x, limit)
  case rand(30)
  when 0..1
    # spawn 3 spike platforms and a normal platform
    [
      SpikePlatform.new(30 + (last_x + rand(limit*2+1) - limit) %340, -20,
false),
      SpikePlatform.new(30 + rand(341), -60, rand(2)%2 == 0),
      SpikePlatform.new(30 + rand(341), -100, true),
      StaticPlatform.new(30 + rand(341), -140)
    ]
  when 2..3
    # spawn a boost platform and a normal platform
    [
      BoostPlatform.new(30 + (last_x + rand(limit*2+1) - limit) %340, -20),
      StaticPlatform.new(30 + rand(341), -70)
    ]
  when 4..5
    # spawn a horizontally moving platform
    [HorizontalMoveablePlatform.new(30 + rand(341), -20)]
  when 7
    # spawn 2 vertically moving platforms and a normal platform
    [
      VerticalMoveablePlatform.new(temp = 30 + (last_x - rand(limit+1)) %270, -
20, 1),
      VerticalMoveablePlatform.new(temp + 70, -158, -1),
      StaticPlatform.new(30 + rand(341), -240)
    ]
  else
    # spawn a normal platform
    [StaticPlatform.new(30 + (last_x + rand(limit*2+1) - limit) %340, -20)]
  end
end
```

2. Generate a random breakable platform at random position above the screen

`generate_random_breakable_platform -> Array[Platform]`

```ruby
# Generate a breakable platform that break on collision with player
def generate_random_breakable_platform
  [BreakablePlatform.new(30 + rand(341), -20)]
end
```

3. Move the platform vertically to scroll platforms down y pixel

`platform_move_y (platform, y)`

```ruby
# Move a platform in y ordinate
# @param platform: the platform to move
# @param y: the distance to move
def platform_move_y(platform, y)
  platform.y -= y
  platform.top = platform.y - platform.h/2
  platform.bottom = platform.y + platform.h/2
end
```

4. Draw a platform on the screen

`platform_draw` (platform)

```ruby
# Draw a platform on screen
# @param platform: the platform to draw
def platform_draw(platform)
  case platform.type
  when :spike
    # Change the image of spike platform after a small delay (if applicable)
    if platform.is_spike and (Gosu.milliseconds - platform.start_delay >
platform.delay_time)
      platform.img = platform.img_active
    elsif not platform.is_spike and (Gosu.milliseconds - platform.start_delay >
platform.delay_time)
      platform.img = platform.img_normal
    end
  when :break
    # Change the image of breakable platform since colliding with player
    if platform.broken == nil
      platform.img = platform.imgs[0]
    elsif Gosu.milliseconds - platform.broken < 50
      platform.img = platform.imgs[1]
    elsif Gosu.milliseconds - platform.broken < 100
      platform.img = platform.imgs[2]
    else
      platform.img = platform.imgs[3]
    end
  end
  platform.img.draw_rot(platform.x, platform.y, ZOrder::PLATFORMS)
end
```

5. Change the state of a single spike platform and play sound effect

`change_state` (spike_platform)

```ruby
# Change the state of a spike platform
# @param spike_platform: the spike platform to change state
def change_state(spike_platform)
  spike_platform.sfx.play(0.3)
  spike_platform.is_spike = !spike_platform.is_spike
  spike_platform.start_delay = Gosu.milliseconds
end
```

6. Change the state of all spike platform in a platforms list

`change_state_platforms` (platforms_list)

```ruby
# Change the state of all spike platforms in a list
# @param platforms_list: the list of platforms
def change_state_platforms(platforms_list)
  platforms_list.each do |platform|
    if platform.type == :spike
      change_state(platform)
    end
  end
end
```

7. Activate a boost platform and play sound effect

active (boost_platform)

```
# Activate a boost platform
# @param boost_platform: the boost platform to activate
def active(boost_platform)
  boost_platform.img = boost_platform.img_active
  boost_platform.sfx.play
end
```

8. Move left and right a horizontal moveable platform

move_horizontal (horizontal_moveable_platform)

```
# Move left and right a moveable platform
# @param moveable_platform: the moveable platform to move
def move_horizontal(moveable_platform)
  if moveable_platform.x < moveable_platform.w/2 or moveable_platform.x >
Window::WIDTH - moveable_platform.w/2
    # Invert direction if at the edge of screen
    moveable_platform.dir = - moveable_platform.dir
  end
  moveable_platform.x += moveable_platform.vx * moveable_platform.dir
  moveable_platform.left = moveable_platform.x - moveable_platform.w/2
  moveable_platform.right = moveable_platform.x + moveable_platform.w/2
end
```

9. Move up and down a moveable platform

move_vertical (vertical_moveable_platform)

```
# Move up and down a moveable platform
# @param moveable_platform: the moveable platform to move
def move_vertical(moveable_platform)
  if Gosu.milliseconds - moveable_platform.t > 2500
    # Invert direction after a certain amount of time
    moveable_platform.dir = - moveable_platform.dir
    moveable_platform.t = Gosu.milliseconds
  end
  moveable_platform.y += moveable_platform.vy * moveable_platform.dir
  moveable_platform.top = moveable_platform.y - moveable_platform.h/2
  moveable_platform.bottom = moveable_platform.y + moveable_platform.h/2
end
```

10. Break a breakable platform on collision with player and play sound effect

platform_break (breakable_platform)

```
def platform_break(breakable_platform)
  if breakable_platform.broken == nil
    breakable_platform.broken = Gosu.milliseconds
    breakable_platform.sfx.play(0.5)
  end
end
```

11. Drop a broken platform

drop (breakable_platform)

```
# Drop a breakable platform after collision with player
# @param breakable_platform: the platform to drop
def drop(breakable_platform)
  breakable_platform.vy += Gravity
  breakable_platform.y += breakable_platform.vy
  breakable_platform.top = breakable_platform.y - breakable_platform.h/2
  breakable_platform.bottom = breakable_platform.y + breakable_platform.h/2
end
```

12. Animate a platform (moving or dropping)

animate_platform (platform)

```ruby
# Animate a platform
# @param platform: the platform to animate
def animate_platform(platform)
  case platform.type
  when :movehori
    move_horizontal(platform)
  when :movevert
    move_vertical(platform)
  when :break
    drop(platform) if platform.broken != nil
  end
end
```

[Button's functions]

1. Check if the mouse coordinates is inside the button's hitbox and change the appearance of the button

mouse_in? (button, mouse_x, mouse_y) **-> boolean**

```ruby
# Check if mouse is hovered over a button and change its appearance
# @param button: the button to check
# @param mouse_x: mouse's x ordinate
# @param mouse_y: mouse's y ordinate
def mouse_in?(button, mouse_x, mouse_y)
  if (button.x - button.w/2 <= mouse_x and mouse_x <= button.x + button.w/2)
and (button.y - button.h/2 <= mouse_y and mouse_y <= button.y + button.h/2)
    button.img = button.pressed
    return true
  else
    button.img = button.normal
    return false
  end
end
```

2. Check if mouse clicks on button

mouse_clicked? (button, mouse_x, mouse_y) **-> boolean**

```ruby
# Check if a button is clicked
# @param button: the button to check
# @param mouse_x: mouse's x ordinate
# @param mouse_y: mouse's y ordinate
def button_clicked?(button, mouse_x, mouse_y)
  if mouse_in?(button, mouse_x, mouse_y) and Gosu.button_down?(Gosu::MS_LEFT)
    return true
  else
    return false
  end
end
```

3. Draw a button

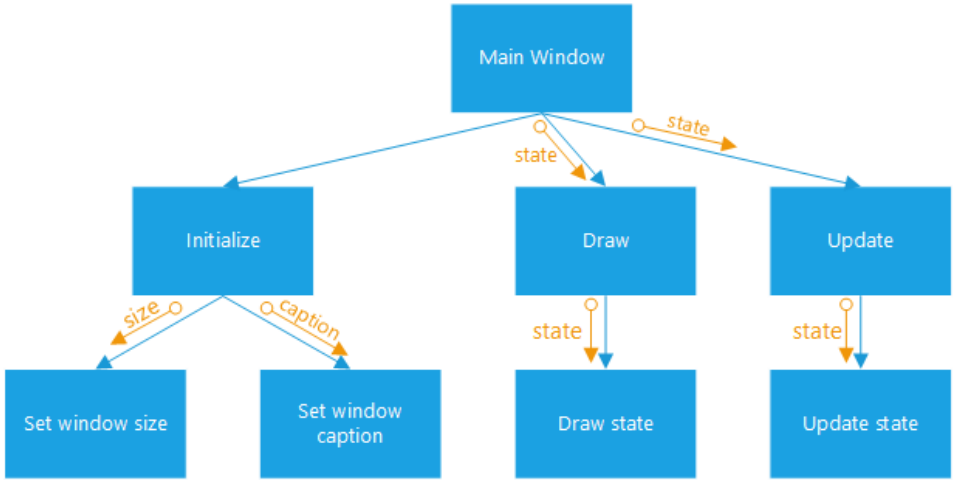
button_draw (button)

```ruby
# Draw a button
# @param button: the button to draw
def button_draw(button)
  button.img.draw_rot(button.x, button.y, ZOrder::UI)
end
```

Structure chart:

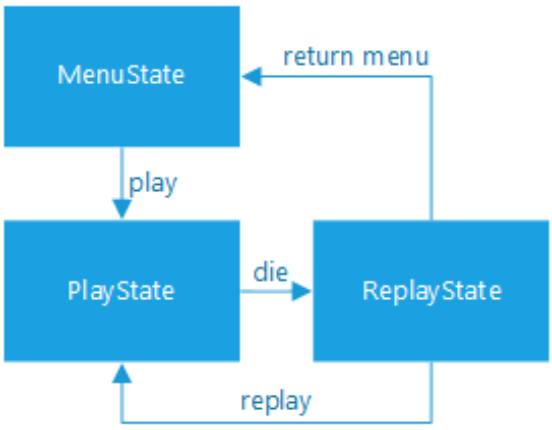
Main Window

This is the window that the game is drawn on. When calling the draw and update method, it will call the draw and update method for the current state



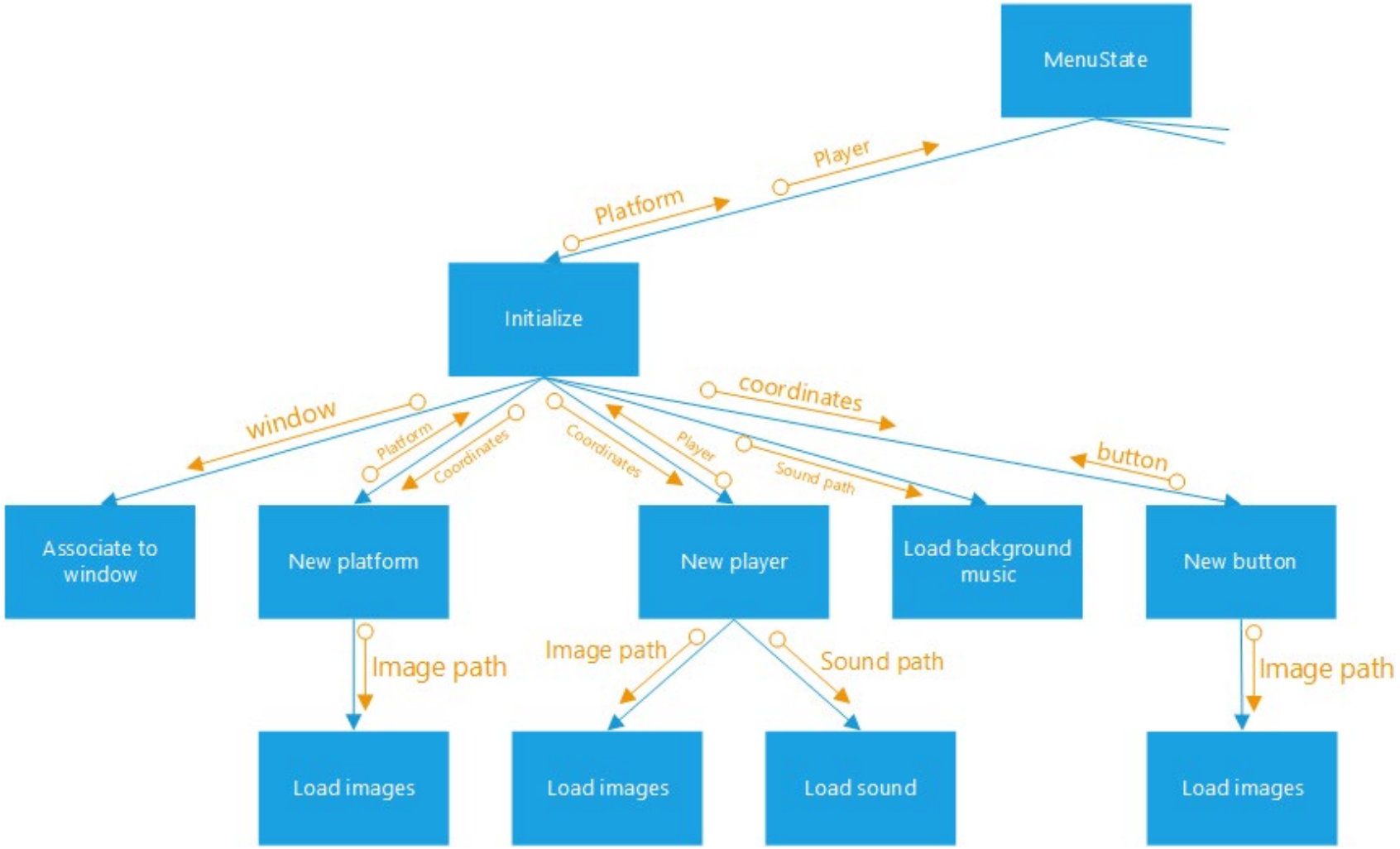
State diagram

There are three separate states, and the game can switch between states.
When open the game, it will enter MenuState, then pressing play button will make it switch to PlayState.
When the player die, the game switches to ReplayState, then the user can choose to replay the game or return to menu.



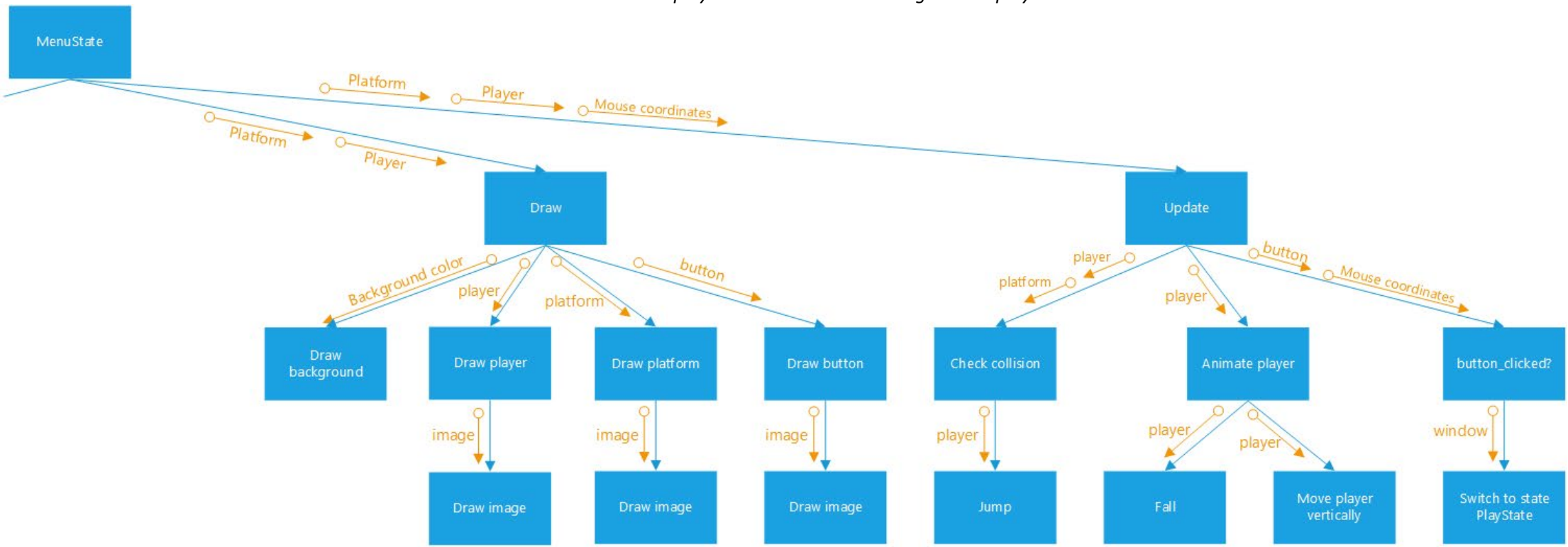
Menu State (1)

In this state, an unplayable player will be created on a platform for decoration.
Background music is loaded and played, and play button is created.



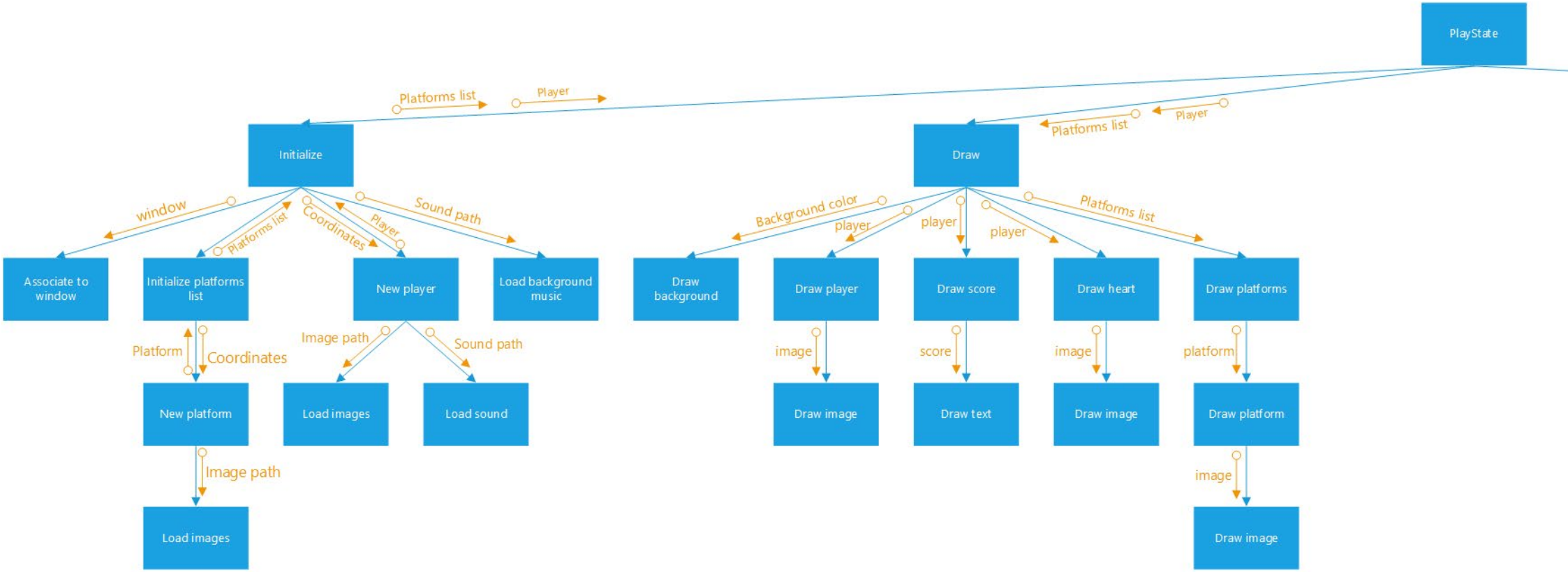
Menu State (2)

Background, player, platform, and button are drawn on the screen. The player is continuously animated.
The user can click on the play button on the screen to go to the play state.



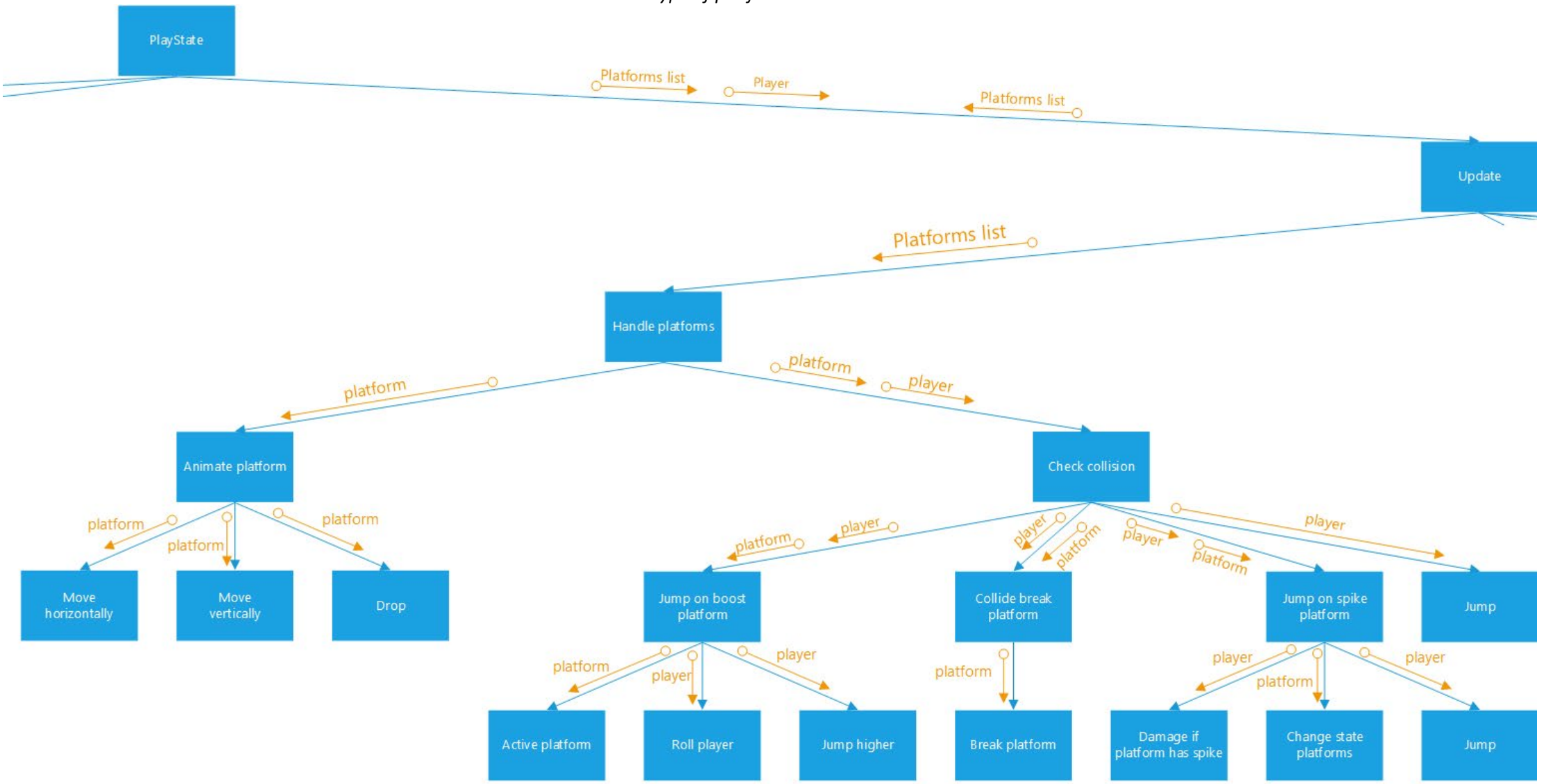
Play State (1)

In play state, a list of platforms will be generated, and a new player will be created on the lowest platform. Background music is loaded and played. Then, the player will be drawn on the screen with the player score and health. All the platforms in the list are also drawn on the screen.



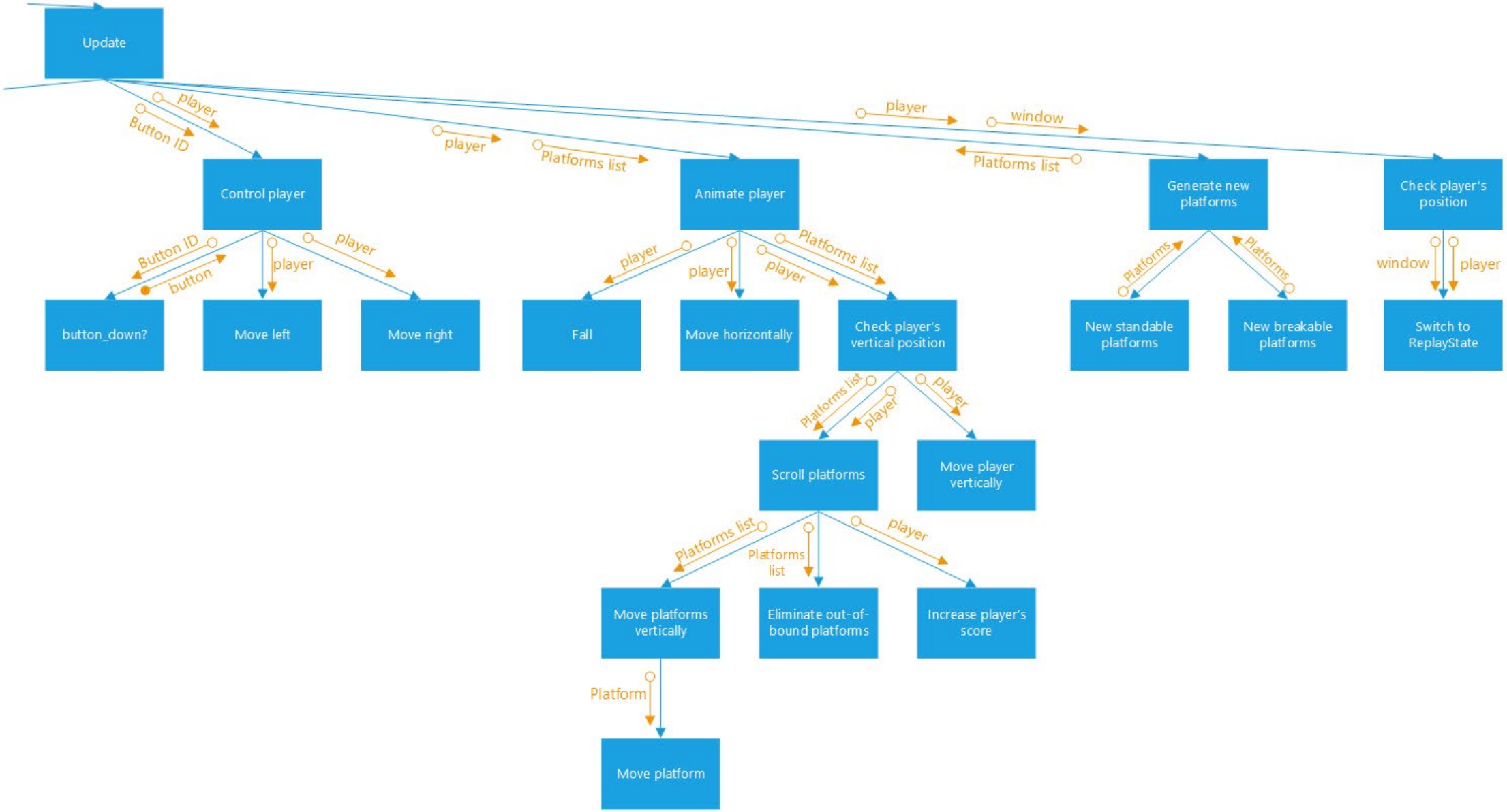
Play State (2)

After that, all the platforms will be handled and animated. Then the player will be checked for collision with each platform in the list and perform different actions base on what type of platform the user collides.



Play State > Update (3)

The player can be controlled with keyboard and be animated so that it will move and fall under gravity.
If the player's height reaches a certain value, instead of move player up, the platforms will be scrolled down, and increase the score.
Platforms that go below the screen are removed, and new platforms will be generated at the top of window randomly.
Finally, if the player falls to the bottom of the window, the game will switch to Replay State



Replay State

In this state, a player will fall from the top to the bottom of the window, indicating that the player has died. Then the old high score is loaded from file to compare with the current score. If the current score is higher, the high score will be updated. Next, two buttons are drawn on screen for user to choose to replay the game or return to menu.