Criterion C: Product Development

Techniques Used

a) File I/O
b) Complex Selection (Nested If - Also Algorithmic Thinking)
c) Breaking Up Code into Classes
d) Extending Classes (Inheritance)
e) Methods with Parameters
f) Methods that Return Values
g) Simple Selection (If/Else)
h) Use of Godot Game Engine
i) Use of External Module

A. File I/O

```
var score_file = "user://new_score.save" #saves highscores

func load_score(): #run when starting, loads high score, times played, and cactus escaped
	var file = File.new()
	if file.file_exists(score_file):
		file.open(score_file, File.READ)
		save_file_data = parse_json(file.get_as_text()) #parse the JSON and make it a dictionary
		#highscore = file.get_var()
		highscore = save_file_data["highscore"]
		times_played_again = save_file_data["times_played_again"]
		cactus_escaped = save_file_data["cactus_escaped"]
		file.close()
	else:
		highscore = 0 #set values if no save file found
		times_played_again = 0
		cactus_escaped = 0
```

The method creates a File object, which can access or store data in a file. If an existing save file is found, it opens up the file in read-only mode and retrieves the variable stored within it, using it to set the highscore, times the game was played, and the amount of cacti jumped over. After doing so, it closes the file. If an existing save was not found, the variables are set to 0.

```
func save_data():
	save_file_data["highscore"] = highscore #save all of the new data into the save_file_data dictionary
	times_played_again = times_played_again + 1
	save_file_data["times_played_again"] = times_played_again
	save_file_data["cactus_escaped"] = cactus_escaped
	var file = File.new()
	file.open(score_file, File.WRITE)
	#file.store_var(save_file_data)
	file.store_line(to_json(save_file_data)) #save the dictionary as json file
	file.close()
```

The method above opens the save file in read and write mode and stores the high score, times the game was played, and the amount of cacti jumped over.

B. Complex Selection (Nested If - Algorithmic Thinking)

```
# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta: float) -> void:
	if !Global.dead and Global.started and !Global.paused:
		if Global.elapsed_time != 0 and Global.elapsed_time >= new_enemy_time: #spawn new cactus every set amount of time
			if Global.elapsed_time >= 75:
				new_enemy_time += 1.5
			else:
				new_enemy_time += (4 - Global.elapsed_time/25)
			create_enemy()
	elif (Global.dead):
		if Input.is_action_just_pressed("ui_focus_next"):
			get_parent().add_child(load("res://src/Ground.tscn").instance())
			queue_free()
			Global.reload()
```

The method above is the _process method. If the player isn't dead, the game has started, and the game isn't paused, it will attempt to make new enemy cacti. The nested if statement checks to see whether or not the time elapsed since the game started is greater than the elapsed time when the game can create a new cactus. If it is, it will choose a new time when the game can create a new cactus, and it will run the command to create a new cactus enemy. Initially, the time between new cacti spawning is long (up to 4 seconds), but it reduces as the game progresses in order to increase the game's difficulty. After the game has been running for 75 seconds, it sets the time between cacti to 1.5 seconds, so that the game doesn't become impossible to beat.

Every frame, it also checks to see whether the player is dead. If it's true, it listens for the user inputting the "R" key. If it's clicked, a new instance of the Ground class (as well as its children - including all visual components in the game) is created. The queue_free method is called, which clears the current Ground class and its children from memory, leaving only the new instance of Ground. Then, the reload method is called, which resets global variables, such as the score. Along with the new instance of Ground, which resets the background and the player, it appears as if the game started anew.

C. Breaking Up Code into Classes

```
extends KinematicBody2D

var velocity = Vector2.ZERO
onready var anim = $AnimationPlayer
const position_x = 0 #starting x position for player

func _physics_process(delta: float) -> void:
	if Global.started and !Global.dead:
		check_jump()
		run_animation()
		apply_gravity(delta)
		if self.global_position.x != position_x: #made to filter for when collisions aren't detected, but players are displaced by enemiy collisions
			Global.die()
	else:
		anim.stop() #stop all animations and make sure screen is frozen once death happens

func check_jump() -> void:
	if Input.is_action_pressed("move_up") and is_on_floor():
		velocity.y -= Global.gravity * 0.75 #change velocity if player wants to jump and is on floor
		GlobalAudioStreamPlayer.play_sound(GlobalAudioStreamPlayer.jump_sound)

func apply_gravity(delta: float) -> void:
	velocity.y += Global.gravity * delta #make sure gravity is applied to the player
	velocity = move_and_slide(velocity, Global.FLOOR_NORMAL)

func run_animation() -> void:
	if is_on_floor(): #run different player animations depending on whether or not you are jumping
		anim.play("Run")
	else:
		anim.play("Jump")
```

I broke the code up into numerous classes: Player, Enemy (all the cacti extend this), Score Label, Starting Screen, Death Screen, AudioStreamPlayer, and Ground. The code above is for the Player class. It will jump into the air if the player clicks the up arrow and the player model is on the ground. It also chooses between playing the "Run" animation (when it's on the ground) and the "Jump" animation if it's off the ground, and it applies gravity to the player.

D. Extending Classes (Inheritance)

```
extends KinematicBody2D
class_name Enemy

var velocity = Vector2.ZERO
var speed = -3 - (Global.elapsed_time / 5.0)

# Called when the node enters the scene tree for the first time.
func _ready() -> void:
	add_to_group("ENEMIES") #add all enemies to the group "ENEMIES" so they can be tracked easily later

func _physics_process(delta: float) -> void:
	if !Global.dead and Global.started:
		velocity.x = speed #ensures constant velocity to the left
		var collision = move_and_collide(velocity) #move and check for collisions
		if collision and collision.collider.name == "Player": #if a collision happens (and it's with the player, not the tilemap), the player dies
			Global.die()

# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta: float) -> void:
	speed = -3 - (Global.elapsed_time / 5.0)
	if self.global_position.x < -250: #delete enemy at this position (once it's entirely out of frame)
		queue_free()
```

The code above is used by all enemies extending the Enemy class, which is all of the cacti in the game. It includes methods to ensure a constant movement towards the left of the screen, the self-deletion of an object if it's greater than 250 pixels out of frame, and collision detection with the player.

E. Methods with Parameters

```
func spawn_enemy(scene: PackedScene) -> void:
	var new_cactus = scene.instance() #instantiate new cactus and put it right outside the screen
	new_cactus.global_position = Vector2(1040, 560)
	if random_num(1, 2) == 1:
		new_cactus.set_scale(Vector2(-1,1)) #flip orientation of cactus
	add_child(new_cactus)

func create_enemy() -> void:
	var random_num = random_num(1, 4)
	if random_num == 1: #spawn large cactus
		spawn_enemy(base_large_cactus)
	elif random_num == 2: #spawn small cactus
		spawn_enemy(base_cactus)
	elif random_num == 3: #spawn triple cactus
		spawn_enemy(base_triple_cactus)
	else: #spawn double cactus
		spawn_enemy(base_double_cactus)
```

In the code above, the create_enemy function calls the spawn_enemy function with one of the four cactus "scenes" (one of the four cactus classes) as an argument. The spawn_enemy function creates a new instance of a cactus for the player to jump over, changes its position to right outside of the screen, and adds it as a child of Ground.

F. Methods that Return Values

```
var rng = RandomNumberGenerator.new() #random number generator
```

```
func random_num(minimum: int, maximum: int) -> int: #returns random number between range (inclusive)
        rng.randomize()
        return rng.randi_range(minimum, maximum)
```

The function in the code above uses an instance of GDScript's built-in RandomNumberGenerator class to generate an integer between the two values (inclusive). Every function call, random_num changes the seed value of the random number generator with the randomize method, and then it returns an integer between the minimum and maximum numbers supplied.

G. Use of Godot Game Engine

I utilized the Godot Engine and the programming language it uses to make my Dinosaur Game. It provided a built-in game loop that would run the methods _process and _physics_process in all of the classes, which would call my user-defined methods to keep the game moving. It also provided a graphical window that I used to see a preview of all of the sprites in my game and how they would all look together on one screen. Moreover, the game engine had built-in methods to apply gravity and detect collisions between character hit-boxes.

H. Use of External Module

```
onready var jump_sound = preload("res://src/Audio/Jump.mp3")
onready var death_sound = preload("res://src/Audio/Game Over.mp3")
onready var startup_sound = preload("res://src/Audio/Startup.mp3")
onready var sound_player = $AudioStreamPlayer

func play_sound(stream: AudioStreamMP3) -> void: #plays audio
        sound_player.stream = stream
        sound_player.stream.set_loop(false) #make sure audio doesn't loop
        sound_player.play()
```

In the code above, I used the AudioStreamPlayer module, which plays audio in 2d spaces. I used it as part of play_sound, a method that any other class could call to play the Mp3 file passed to the method.

Word Count: 770