Name: Gayathri Poluri

Class Name: CPSC 5270 Graphics/Game Project (25 SQ)

Git Hub Link: https://github.com/gayathripoluri/Game-project.git

Prince’s Pursuit

Summary:

Prince’s Pursuit is a 2D Game where he doesn’t run aimlessly but to prove his true love to his enchantress (Princess). To reach his queen and prove his love, he needs to fight the odds the universe throws at them, so the princess runs in a forest, and what? Forest doesn’t welcome him, but gives him harsh treatment, where the cursed animals try to attack him(sprites like wolves, bears), he can counterattack them using magic bolts. Initially, the magic bolts are 5, and during his run, he will get a few more, and every time he kills an animal, he can be rewarded with a gem or a life, or possibly nothing. The terrible weather (fog, rain) in the middle of the run interrupts him. (CPUParticles2D)

When life throws its tantrums, the universe has a way of rewarding. He will get rewarded with a magical tree in one out of n runs. The magical tree can randomly give 3 gems at a time or slow down time. The Interesting Part of the game is that the prince needs to reach the end of the run(i.e, reach the princess) in a limited time with a certain amount of gems that he collects during his run to win the love of the Princess (“No Gems -No Love Policy”)

I'm using an iterative development process in Godot Engine to accomplish the goals of "Prince's Pursuit," beginning with a minimal prototype to establish basic mechanics like movement, jumping, and background scrolling, as seen in the current alpha with main.tscn, player.tscn, and world.tscn. Then, I'm gradually adding features like enemy spawning using a GemSpawner node using a Factory Method pattern, weather effects via CPUParticles2D for fog and rain, and a TimerManager for the 180-second limit, which will be extended by 5 seconds per gem collected. I will employ modular design principles like the Composite pattern for layered visuals, implement a TimerBar and GemCounter on the HUD using an Observer pattern to update dynamically, and source pixel-art assets from OpenGameArt.org to maintain a consistent visual style. I'll test each addition to ensure scalability and balance gameplay and narrative progression toward the princess's jaw-dropping conclusion.

Primary Aesthetics

The three main aesthetics guiding *Prince's Pursuit* are **Sensation**, **Fantasy**, and **Narrative**. While all three are part of the game's design vision, the **Alpha version** focuses primarily on **Sensation** and **Narrative**, laying the emotional and visual foundation of the experience.

1)Sensation:

The game aims to create a vivid, emotionally engaging forest setting, one that *feels alive*. The prince moves through an enchanted woodland where movement is fluid, the environment shifts dynamically, and each action feels responsive. In the current implementation, much of this is already visible:

- The **player's movement** is handled smoothly through the CharacterBody2D node using move_and_slide() with a speed of 300.0. Direction is set via Input.get_axis("ui_left", "ui_right"), and deceleration uses move_toward() to create a natural slowdown.
- The **sprite flips** automatically (flip_h = true or false) based on direction, so the prince always faces the way he's running, adding intuitive feedback.
- Animations are controlled through **AnimationPlayer**, transitioning between "run", "jump", "fall", and "Idle" based on player state. For example:
    - "jump" plays when jumping (is_on_floor() check)
    - "fall" plays when descending
    - "run" when the prince is moving on the ground
    - "Idle" when he's standing still

The **background parallax effect** uses ParallaxBackground, scrolling smoothly at 100 pixels/sec with Forest.png, adding depth and a sense of motion to the forest.

These elements already bring life to the scene. Features like **CPUParticles2D** for fog/wind, ambient sound, gem glow, and a fully interactive HUD are in progress for the next milestone.

2)Narrative:

The story behind *Prince's Pursuit* adds emotional weight to the gameplay. This isn't just an endless runner — it's a race of love. The prince is on a time-bound quest to collect gems and break the enchantment keeping his princess out of reach. If he fails, she disappears forever. If he succeeds, his devotion is proven.

While full narrative elements (like gems, time limits, and cutscenes with the princess) are still in development, the current alpha already hints at this journey:

- The **title screen animation** in bg.tscn introduces the theme of the pursuit.

- The **forest scrolls** in the background, creating a mythical atmosphere and evoking the feeling of a long, determined journey.
- The **player's determined movements**, captured through jump and run animations, reflect his urgency and purpose.

Primary Mechanics:

The foundation for the platforming experience has been laid by my implementation of the four main mechanics: movement, jumping, animation switching, and background scrolling.

Movement:

The prince can move across the forest floor thanks to the Movement mechanic, which is consistent with the original idea of navigating a cursed forest. Player input is captured in player.gd using Input.get_axis("ui_left", "ui_right"), which returns a value of -1 for left, 1 for right, or 0 for no input. The prince's horizontal velocity is determined by this value, which is stored in a direction variable. The formula for calculating velocity is velocity.x = direction * SPEED, where SPEED is a constant set to 300.0. This indicates that the prince is moving in the selected direction at 300 pixels per second. To avoid sudden stops and guarantee steady travel, I use velocity.x = move_toward(velocity.x, 0, SPEED) to smoothly slow the prince to a stop when no input is detected (direction = 0). This deceleration makes sure that the prince's speed, measured in velocity, gradually drops to zero over a short period of time if he is heading to the right at, say, 300 pixels per second and the player lets go of the key. Until it hits zero, x -= SPEED * delta per frame.

To apply movement, I used Godot's move_and_slide() function on the CharacterBody2D node inside player.tscn. This function lets the prince move smoothly across the ground while also handling collisions automatically. For example, if he walks into a wall (planned for later), instead of getting stuck, move_and_slide() will make him slide along the wall edge, giving a realistic physics feel.

To enhance the visual feedback, I made the sprite flip depending on the direction:

- When the prince moves left (direction = -1), flip_h is set to true
- When moving right (direction = 1), flip_h is set to false

This small touch helps the player feel more in control, as the prince visibly reacts to their input.

Jumping:

Jumping is an essential mechanic to navigate the forest, like hopping over roots or avoiding holes (to be added soon). I used Input.is_action_just_pressed("ui_accept") to detect when the spacebar is pressed and is_on_floor() to ensure he jumps only when touching the

ground. If both conditions are met, I assign velocity.y = JUMP_VELOCITY, with the constant set to -400.0 to push him upwards (since Godot's Y-axis points downwards).

Gravity is applied naturally as the prince ascends and falls. I used:

var gravity = ProjectSettings.get_setting("physics/2d/default_gravity")

Then inside _physics_process(delta), gravity is applied like this:

if not is_on_floor():

velocity.y += gravity * delta

This makes sure the jump arc feels realistic and smooth, no matter the frame rate. For example, with gravity set at 980, the prince falls by around 16.33 pixels every frame at 60 FPS.

To make this more immersive, I used AnimationPlayer to switch animations:

- jump plays when the prince leaves the ground
- fall triggers once his velocity turns positive (i.e., he's descending)

This gives the player visual confirmation of the jump state, enhancing the overall feel.

Animation Switching

The animation system keeps the prince visually in sync with his movement and actions. I used an AnimationPlayer node (referenced as anim = get_node("AnimationPlayer")) to switch between different states.

In _physics_process(delta):

- If the prince is moving horizontally, anim.play("run") is triggered
- If jumping, anim.play("jump")
- When falling (velocity.y > 0), it switches to anim.play("fall")
- If standing still (velocity.x = 0 and velocity.y = 0), it goes back to anim.play("Idle")

All animation sequences (run, jump, fall, idle) are handled through the AnimatedSprite2D node. These changes happen every frame, so transitions look smooth and responsive. This not only adds polish but helps the player understand what the prince is doing just by looking at him.

Background Scrolling

To make the forest feel like it’s alive and constantly moving, I added a parallax scrolling background using ParallaxBackground in world.tscn. The script controlling this sets a scrolling_speed = 100. Inside _process(delta), I update the background like this:

scroll_offset.x -= scrolling_speed * delta

This scrolls the background smoothly from right to left, giving the illusion that the prince is running through a large forest. It works independently of frame rate because of delta.

The ParallaxLayer holds a Sprite2D node with the texture Forest.png (from OpenGameArt.org). With motion_scale = 1.0, it scrolls at the same speed as set. Though the TileMap ground stays in place relative to the player, this background adds depth and makes the world feel more dynamic. In the future, this will also sync well with enemies, weather effects, and collectibles.

Alpha Reflection

Since my initial checkpoint, I’ve made solid progress toward the original vision of "Prince’s Pursuit," a 2D platformer set in a cursed forest. The core gameplay — movement, jumping, animation, and background scrolling — is working.

One important shift was switching from KinematicBody2D to CharacterBody2D, which works better with Godot 4's physics system. It also simplified the logic since move_and_slide() takes care of ground collisions (instead of manually coding checks).

I’ve also added a **main menu** using main.tscn with **Play** and **Quit** buttons.

Pressing Play calls: **get_tree().change_scene_to_file("res://world.tscn")**

And Quit calls**: get_tree().quit()**

This made the game more polished and user-friendly.

I spent time fine-tuning the movement:

- SPEED = 300.0 felt responsive
- Originally tried 500.0 but it was too fast

Animations took careful tweaking. For example, if I didn't check velocity.y > 0 correctly, the fall animation might interrupt the jump or idle ones. Getting that right made the transitions much smoother.

Also, learning how to implement scrolling with ParallaxBackground helped me understand frame-rate independence using delta. Overall, everything is shaping up well, and I'm on track to add enemies, weather, and gem collectibles in the next phase.

Software Elements:

Application of Creational Programming Pattern: *Singleton*

In main.gd, I used a Singleton-like approach by relying on Godot's SceneTree system through get_tree(). This gave me one central point to control the game's flow, such as when the player clicks "Play" or "Quit" in the menu scene (main.tscn). Specifically:

- **get_tree().change_scene_to_file("res://world.tscn")** starts the game.
- **get_tree().quit()** exits the game entirely.

Keeping these controls in one place makes things much cleaner. Instead of having scene-changing logic scattered across multiple scripts, everything is handled from one file. That means fewer bugs, fewer chances of loading the wrong scene, and much easier debugging.

If I didn't use this setup, it could break the flow of the game — the menu might never unload, or the game might freeze if scenes aren't properly managed. Using Godot's built-in singleton behavior through SceneTree keeps transitions smooth and scalable. I can easily expand this later by adding settings, a pause menu, or other screens without changing much of the existing logic.

Application of Structural Programming Pattern: *Composite*

I used the Composite pattern in world.tscn through the ParallaxBackground system. It contains a ParallaxLayer, which in turn holds a Sprite2D showing a forest image (Back.png). These elements work together as one unit, scrolling the background to create depth and movement.

In the script, I added:

scroll_offset.x -= scrolling_speed * delta

with scrolling_speed = 100, so the background scrolls to the left. Thanks to delta, it scrolls smoothly no matter the frame rate — about 1.67 pixels per frame at 60 FPS.

This scrolling background makes the forest feel alive. Without it, the game would look flat and static, even though the player is moving. The Composite pattern here lets me control all background layers as one, and if I want to add more layers (like clouds or mountains), I can just drop them in and assign different scroll speeds using motion_scale.

Application of Behavioral Programming Pattern: *Observer*

In player.tscn, the AnimationPlayer acts like an Observer, keeping an eye on the prince's state. It watches values like velocity and player input from player.gd, then updates the animations based on those changes.

Here's how it reacts:

- run plays when the player is moving left/right.
- jump triggers when the jump button (space) is pressed and the prince is on the ground.
- fall starts once the prince is in the air and begins descending (velocity.y > 0).
- Idle plays when the prince is standing still.

All of this is checked every frame in _physics_process(delta), keeping the prince's visuals perfectly in sync with what he's doing. Without this setup, the prince would just stay in one pose — the game would feel stiff and lifeless. But thanks to this pattern, the animations respond instantly to gameplay, making everything feel fluid and immersive.

And the best part? Because AnimationPlayer reacts to state changes instead of controlling them, the animation system stays separate from the movement code — meaning I can improve or change one without messing up the other.

Application of Game Engine Technology: *CharacterBody2D*

For movement and physics, I used a CharacterBody2D node for the prince in player.tscn. This is where all the movement logic lives — running, jumping, and falling.

The function move_and_slide() handles it all:

velocity.x = direction * SPEED

velocity.y += gravity * delta

This means the prince accelerates downward naturally due to gravity, and collides properly with the ground (which is part of the TileMap in world.tscn). If I had to write custom collision code for this, it would've taken way more time and likely led to bugs like falling through the floor or getting stuck on walls.

Using Godot's built-in physics means the prince interacts with the environment just like any other object would — bouncing off the ground, sliding when needed, and stopping when hitting obstacles. It makes platforming feel responsive and smooth, and it also saves me from reinventing the wheel.

Application of Game Engine Technology: *AnimationPlayer*

The AnimationPlayer node in player.tscn is the heart of the prince's animations. It handles transitions between all the major states — idle, run, jump, and fall.

Inside player.gd, I trigger animations with:

anim.play("run")

anim.play("jump")

anim.play("fall")

anim.play("Idle")

Each animation is a sequence of sprite frames set up in the Godot animation editor using AnimatedSprite2D. For example, the running animation loops at 10 FPS to create a smooth running motion, while jump and fall are quick, single-frame transitions that match the prince's movement.

Godot's timeline-based animation editor also helps blend transitions between these states. So if the prince jumps and then falls, the animations switch automatically without snapping — everything feels natural.

If this system didn't exist, I'd have to write frame-by-frame logic manually, which would take longer and probably look worse. AnimationPlayer keeps the visuals tight and gives immediate feedback to the player, making it a vital part of the platforming experience.

Current Scene Structure

main.tscn (Node2D)

- **Play (Button):** changes to world.tscn
- **Quit (Button):** exits the game

world.tscn (Node2D)

- **TileMap:** holds ground tiles (with collision)
    - **GroundLayer:** painted ground tiles (z_index = 0)
- **ParallaxBackground:**
    - **ParallaxLayer:**
        - **Sprite2D:** uses Forest.png as background
- **Player:** instance of player.tscn

player.tscn (CharacterBody2D)

- **AnimatedSprite2D:** handles run, jump, fall, idle
- **CollisionShape2D:** for physics interactions
- **AnimationPlayer:** controls animation transitions

.