Technical Design Document: Body Snatcher

Team Delta

Elias El Khoury, Haifa Al Ashkar, Nael Haidar

Scripts

Player MonoBehaviour Scripts

PlayerMovement.cs:

The PlayerMovement.cs script controls the basic movement and rotation of the player character. It utilizes straightforward methods to ensure responsive and intuitive gameplay. Key functionalities include the following functions:

- Move(): Implement movement of the character in the desired direction based on player input using Vertical axis. It also implements sprinting by increasing translation speed when pressing Left or Right Shift.
- Rotate(): Rotate smoothly the character, ensuring alignment with the movement direction or a specified target using Horizontal Axis.

PlayerCombatSystem.cs:

The PlayerCombatSystem.cs script manages the combat mechanics for the player character. It primarily focuses on enabling the player to attack guards within a specific range. Key features include:

- GetClosestGuard(): Determine the nearest guard for combat interactions. It iterates through all guards in the scene, calculating the distance between the player and each guard using Vector3.Distance. The function updates

the closest guard reference whenever a shorter distance is detected, ensuring the most relevant target is selected for combat. If no guards are found, the function returns null. This ensures efficient and accurate target acquisition.

- FindClosestGuardComponentsToAttack(): Triggered by an animation event during the player's attack sequence, its primary purpose is to identify the nearest guard and initialize components necessary for executing the attack: need the transform and health systems of the guard. Also, It updates the _attackOnGoing Boolean which is used by the guard's FSM to detect the attack and shield himself.
- Attack(): Triggered by animation event during the player's attack sequence, it aims to look or face the guard getting attacked and damage him if the player is within the attack range.
- AttackOngoing(): Triggered by animation event during the player's attack sequence, it aims at updating the _attackOnGoing Boolean used by the guard for shielding.
- ResetAttackOngoingBool(): It resets the _attackOnGoing bool. It is invoked repeatedly every 0.2 seconds.

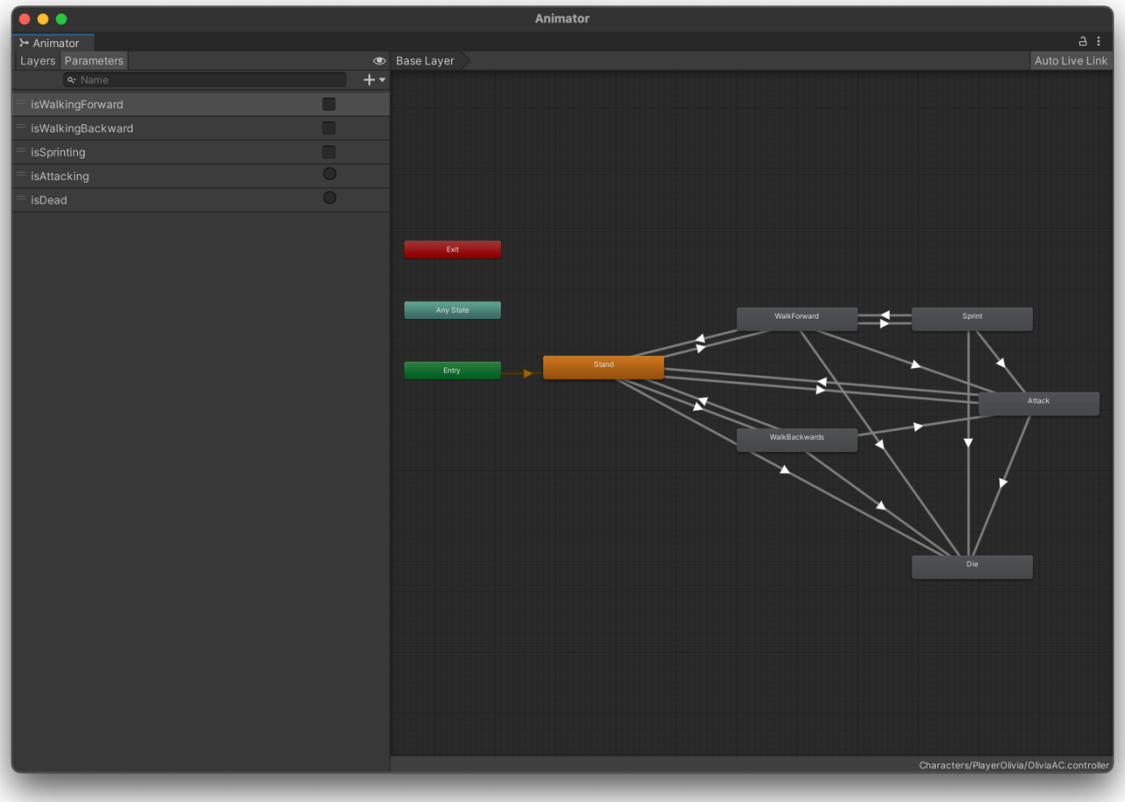
PlayerHealthSystem.cs:

The PlayerHealthSystem.cs script manages the health mechanics for the player character. The main functions implemented are:

- TakeDamage(float damage): When called, it impacts damage on the player's current health. It also calls CheckForDeath().
- Health(float regen): When called, it increases current health of the player by regen amount. It also checks if current health exceeds the maximum allowed health to fix that.

- CheckForDeath(): If the player's current health is less than or equal to zero, it deactivates the player's movement script and health system. It also returns a Boolean depending on whether the current health is still positive or not.

Player StateMachineBehaviour Scripts



PlayerBase.cs:

The PlayerBase class serves as a foundational script for managing player states in the game, extending Unity's StateMachineBehaviour to facilitate interaction with the animation state machine. It provides shared functionality and references to critical player components.

Within this script, there is the initialization of protected references to gameObjects of the player, such as the movement and health scripts, and string animator parameters used for animations.

There is also a definition of several functions:

- Moving(): returns true when the player is moving and false when he's not.
- MovingForward(): returns true when vertical axis is positive (moving forward).
- SprintEnabled(): returns true when either shift keys are pressed.
- CheckForAttackTrigger(): triggers animator attack state when space bar is pressed.
- CheckForDeathTrigger(): triggers animator die state when the player's CheckForDeath() function from the PlayerHealthSystem.cs returns true.

StandState.cs:

This class inherits from the player base class. It also checks at every update for the death and attack triggers defined in PlayerBase.

It also checks for whether the movement state is to be activated or not using the following defined function:

- CheckForMovementState(Animator animator): This function sets the animator's bool value to true for either the walk-forward or walk-backward. It checks for the output bool of the function Moving() and MovingForward().

SprintState.cs:

This class inherits from the player base class. It also checks at every update for the death and attack triggers defined in PlayerBase.

It also checks for when the SprintEnabled() or Moving() functions are false in order to set the animator's sprint Boolean to false, thus exiting this state.

WalkForwardState.cs:

This class inherits from the player base class. It also checks at every update for the death and attack triggers defined in PlayerBase.
It checks for when MovingForward() returns false to set the animator's walk-forward bool to false to deactivate the state.
It checks for when SprintEnabled() returns true to set the animator's sprint bool to true and activate the corresponding state.

WalkBackwardState.cs:
This class inherits from the player base class. It also checks at every update for the death and attack triggers defined in PlayerBase.
It sets the animator's walk-backward bool to false to exit this state when Moving() returns false or MovingForward() returns true.

AttackState.cs:
This class inherits from the player base class. It also checks at every update for the death trigger defined in PlayerBase.
At the state enter, the player's movement script is disabled. In contrast, it enables it back.

Guard MonoBehaviour Scripts

GuardMovement.cs:
The GuardMovement.cs script controls the patrolling and chasing systems of the guard.

1. **Patrolling:**

- StartPatrolling(): Enable the guard's nav mesh agent by setting isStopped to false. It then calls for GoToNextWaypoint().

- GoToNextWaypoint(): Set the destination of the guard to a random waypoint from the Waypoints array. It also sets the agent's speed to that of the patrol speed float variable.
- ReachedDestination(): Return true when the agent's remaining distance is less than or equal to _wpStoppingDistance float variable which means it reached the waypoint destination.

2. **Chasing:**

- ChaseTriggered(): Check for the output of the PlayerDetected() function from the PlayerDetectionConeVision which corresponds to whether the player is within the cone vision of the enemy. This function returns true when the player is detected.
- StartChasing(): Enable the guard's nav mesh agent by setting isStopped to false. It then sets the agent's destination to that of the player and sets its speed to that of chasing.
- UpdatePlayerLocation(): It updates the agent's destination to the new one of the player.
- ReachedPlayer(): Return true when the agent's remaining distance is less than or equal to _playerAttackRange float variable which means it reached the player after stopping the agent.

GuardCombatSystem.cs:

The GuardCombatSystem.cs script manages the combat mechanics for the guard character. Key functions include:

- FacePlayer(): Uses LookAt function on the guard's transform to face the player's position.
- PlayerWithinAttackRange(): returns bool positive if the distance between the guard's position and that of the player is less than or equal to a specified _attackRange float value.

- Attack(): Triggered via animation event on the guard's attack animation, it calls the TakeDamage function from the player's health system script to impact the player's health with a damage of value _damageAmount float.

GuardHealthSystem.cs:

The GuardHealthSystem.cs script manages the health mechanics for the guard character. The main functions implemented are:

- TakeDamage(float damage): When called, it impacts damage on the guard's current health. It also calls CheckForDeath().
- CheckForDeath(): If the guard's current health is less than or equal to zero, it deactivates the guard's navmesh agent component and destroys the guard after _countdownTillDestroyed seconds before returning true.

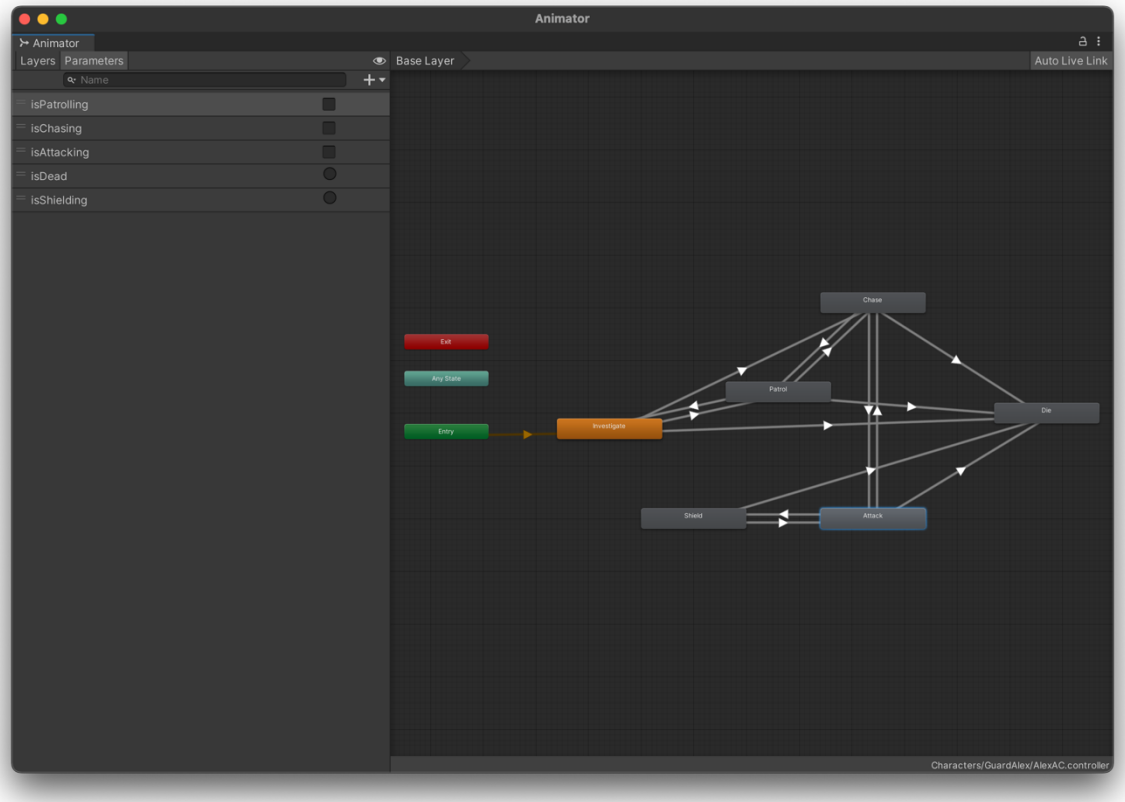
PlayerDetectionConeVision.cs:

It implements a cone-based vision detection system for the guard. It is responsible for identifying objects or characters within a specific field of view (FOV) and returns true when the tag of the gameObject detected is that of the player.

Key functionalities include:

- Define Field of View: The script uses parameters such as viewAngle and viewDistance to define the detection cone. The viewAngle determines the width of the cone, while the viewDistance defines its reach.
- Detection Mechanism: Objects within the cone are detected based on their position relative to the origin of the cone. This is calculated using Vector3.Angle to measure the angle between the forward direction of the entity and the direction to the target.
- Line of Sight Verification: Ensures that detected objects are not obstructed by obstacles using raycasting. This prevents false detections behind walls or other barriers.

Guard StateMachineBehaviour Scripts



GuardBase.cs:

The GuardBase class serves as a foundational script for managing guard's states in the game, extending Unity's StateMachineBehaviour to facilitate interaction with the animation state machine. It provides shared functionality and references to critical player components.

Within this script, there is the initialization of protected references to gameObjects of the player, such as the movement and health scripts, and string animator parameters used for animations. Also, the following function is defined:

- CheckForDeathTrigger(): triggers animator die state when the guard's CheckForDeath() function from the GuardHealthSystem.cs returns true.

InvestigateStateG.cs:

This class inherits from the guard base class. It also checks at every update for the death trigger defined in GuardBase.

It checks for the _gameStart bool which is true only when first starting the game. This is used to launch the guards in patrolling state at the start of the game. Otherwise, idleTime is computed so that after being in the investigation state for _idleTimeThreshold seconds, patrolling starts.
It also checks whether ChaseTriggered() from the guard's movement script is true in order to set the animator's chase bool value to true and to start the chase state.

PatrolStateG.cs:
This class inherits from the guard base class. It also checks at every update for the death trigger defined in GuardBase.
On state enter, the guard calls for StartPatrolling() from his movement script.
Then, on every state update, it checks for the output Boolean value of ChaseTriggered() and ReachedDestination() in order to transition or not to the chase state and to the investigation respectively.
On state exit, the animator's patrol bool is set to false.

ChaseStateG.cs:
This class inherits from the guard base class. It also checks at every update for the death trigger defined in GuardBase.
On state enter, the guard calls for StartChasing() from his movement script.
On state update, UpdatePlayerLocation() updates the player's location for the guard to keep chasing. Then, it checks for the output of ReachedPlayer() and ChaseTriggered to possibly transition to attack state or out of the chase state respectively.

AttackStateG.cs:
This class inherits from the guard base class. It also checks at every update for the death trigger defined in GuardBase.

Also, at every state update, the guard's position is faced to the player's using FacePlayer(). It then checks for the bool values of PlayerWithinAttackRange() or ChaseTriggered() to exit the attack state and go back to chasing or patrolling. It also checks for AttackOngoing() from the player's combat system script to transition to the shield state.

ShieldStateG.cs:
This class inherits from the guard base class. It also checks at every update for the death trigger defined in GuardBase.
On state exit, it sets the animator's attack bool to true.

ContextMenu.cs:

Manages the context menu UI for interacting with items in the inventory.

UIManager.cs:

Handles switching between different UI states, such as inventory, pause menu, and game over.

Inventory.cs:

Manages the player's inventory, including adding, removing, saving, and loading items.

CollectableItem.cs:

Defines behavior for collectible items in the game world, such as adding them to the inventory.

GameSettingsLoader.cs:

Loads and applies game settings, such as audio configurations.

InputManager.cs:

Handles player input for toggling menus and interacting with gameplay elements.

MapToggleManager.cs:

Manages the toggling and display of the in-game map UI.

MapItem.cs:

Defines a specific item type related to map functionality, such as revealing hidden areas.

EnemyHealth.cs: Tracks and manages enemy health, handling damage, death animations, and object destruction.

HealingPotion.cs:

Defines a healing potion that restores health to the player when used. Highlights customization for heal logic.

InventorySlot.cs:

Represents a slot in the player's inventory to manage individual items and their counts.

QuitButton.cs:

Implements logic for quitting the game, ensuring proper handling in both editor and build modes.

ResumeButton.cs:

Handles resuming the game by interacting with the pause menu UI and game state.

SaveManager.cs:

Manages saving and loading of game data, including player progress, inventory, and NPC states.

SaveMenuDropdown.cs:

Populates a dropdown menu for selecting save files and allows loading the selected save.

SettingsManager.cs:

Handles application settings, such as volume and resolution, with functionality for saving and loading preferences.

WinningTrigger.cs:

Implements logic for triggering the winning state, including enabling the winning menu and pausing the game.

GameController.cs:

Manages overall game state, including saving and loading data, and interacting with NPCs and inventory.

NPC.cs:

Represents NPCs in the game, associating them with unique IDs for saving/loading, and integrates with enemy health logic.

ItemData.cs:

Defines a base class for game items, allowing inheritance for specific item types like potions or keys.

InventoryUI.cs:

Manages the display of inventory items in the UI, refreshing and updating item slots dynamically.

QuitConfirmationManager.cs:

Handles the logic for the quit confirmation dialog, ensuring the player can confirm or cancel exiting the game.

GameData.cs:

Stores the serialized state of the game, including player, inventory, and NPC data, used for saving and loading.

SaveMenuController.cs:

Manages the save menu logic, allowing players to input a filename and save their progress.

SliderInitializer.cs:

Initializes a volume slider with the saved music volume setting from PlayerPrefs.

NPCData.cs:

Represents serialized data for NPCs, including their ID, health, position, and rotation, for saving and loading game state.

Environment related scripts:

BackgroundMusic.cs: Manages persistent background music across scenes, ensuring only one instance plays at a time.

AlarmSound.cs: Controls an alarm sound triggered by proximity, allowing interaction to stop the alarm.

LaserPointer.cs: Handles a laser's blinking effect, raycasting, and player damage when hit.

FlashingLight.cs: Creates a flashing light effect by toggling its visibility at specified intervals.

terrainhider.cs: Makes a terrain and its foliage invisible, useful for optimizing or altering visual effects.

OpenDoorWithEvent.cs: Manages door opening and closing actions triggered by player interaction.

Sources

Mixamo

You can use these keywords to search for the corresponding character and animations

- Character: Olivia, Chad, Gas Mask
- Animations:
    - For Olivia: Standing W/ Briefcase Idle, Standing Walk Forward, Standing Walk Backwards, Run With Sword, Standing Melee Attack, Two Handed Sword Death
    - For Gas Mask: Idle, Sneaky Walk, Running, Falling Back Death, Standing Block React Large, Stable Sword Outward Slash
    - For Chad: Laying Severe Cough, Laying Breathless, Laying Moaning

Youtube

- Cinemachine freelook camera tutorial: https://youtu.be/4HpC--2iowE?si=jL9pNwoy5Br_820c
- Sliding door sound: https://youtu.be/Xw9-MRxdZGY?si=2mCIULQqCbWaVKoN
- Alarm sound: https://youtu.be/17EEOQdBwVM?si=n72MbPcPLjsdRaDX
- Laser Beam sound: https://youtu.be/xBGtk3lt_Ug?si=nL6jI-Ob-8iZEQif
- Post-process tutorial: https://www.youtube.com/watch?v=-RBwPD6NNms&pp=ygUbYmxvb20gcG9zdCBwcm9jZXNzaW5nIHVuaXR5
- Cone Vision: https://youtu.be/luLrhoTZYD8?si=EpANZt6iXP8mZhc5
    - PS: The script was heavily modified by me to implement player detection

ChatGPT

ChatGPT was mainly used for debugging syntax