NEA project screenshots

A screen shot of a computer program

AI-generated content may be incorrect.Mouse looking code

A screen shot of a computer

AI-generated content may be incorrect.

Updated mouse look sensitivity and clamps as originals were too fast and high/low

**Module: MouseMovement.cs**

**Description**

MouseMovement handles the player camera rotation (looking up/down and left/right) based on mouse input. It reads the mouse X/Y axes each frame, scales them by a sensitivity value and Time.deltaTime, clamps the vertical rotation to prevent the camera flipping, and applies the rotation to the GameObject’s local transform. The cursor is locked to the screen centre on start.

MouseMovement handles the first person camera for the player. It rotates the camera based on mouse input and clamps the vertical rotation to prevent the camera from flipping, locking the cursor to the centre at the start.I have chosen to do this first as it is a core gameplay element and the program wont be testable without it.

**Justification (why it’s in Stage 1)**

* **Core gameplay requirement:** Precise aiming and looking are fundamental to any FPS. Without camera control the game cannot be tested or played.
* **Testability:** Implementing this first allows early testing of aim-related mechanics (shooting/reloading) and helps tune sensitivity before more complex systems are added.
* **Computational suitability:** This uses real-time input sampling and simple numeric transforms — tasks that are ideal for a computational approach (fast, consistent, frame-based updates).

**Key variables / Data dictionary**

| **Variable** | **Type** | **Purpose / Notes** | **Preset / Valid ranges** |
| --- | --- | --- | --- |
| mouseSensitivity | float | Multiplies mouse input to adjust responsiveness. | Default 470f. Suggested usable range 50–1000. |
| xRotation | float | Stores current vertical (pitch) rotation in degrees. | N/A |
| yRotation | float | Stores current horizontal (yaw) rotation in degrees. | N/A |
| topClamp | float | Minimum pitch (look up) clamp in degrees. | -75f |
| bottomClamp | float | Maximum pitch (look down) clamp in degrees. | 75f |

**Initial Test Plan (what to test for Stage 1)**

* **Functional tests**
  + Move mouse left/right → camera yaw rotates smoothly.
  + Move mouse up/down → camera pitch rotates and is clamped at ±75° (no flipping).
  + Cursor is locked on game start (cursor invisible/locked).
* **Edge / invalid tests**
  + Set mouseSensitivity to very low/high values to ensure no NaN/overflow.
  + Toggle mouse input (disconnect) to ensure no crashes.
* **Performance**
  + Ensure rotation remains smooth at 30+ FPS and 60+ FPS (no stutter).

**Success Criteria (measurable)**

* Camera rotates in response to mouse input; pitch clamped to the range [-75°, 75°].
* No visible camera flipping or sudden jumps during movement.
* Cursor remains locked on start (so mouse input is captured).

**Known limitations / improvements**

* mouseSensitivity default is very high (470) — provide an options slider to allow players to tune it.
* Consider separating camera yaw onto the player body (so character rotates) and pitch to camera only (if not already done).
* Add smoothing or acceleration curve if aiming feels too harsh.

**Justification — MouseMovement (camera control)**

**What it does (brief):** reads mouse X/Y, scales by sensitivity and Time.deltaTime, clamps pitch, applies rotation.

**Why it was coded this way**

* **Separation of concerns / clarity** — The camera rotation is its own script so it can be changed or reused without touching player movement. That follows good decomposition and makes debugging easier (if the camera is weird it’s in one place).
* **Frame-rate independence** — Multiplying input by Time.deltaTime makes movement stable across different frame rates. This avoids sensitivity changing on slow / fast machines and is essential for fair testing and consistent gameplay.
* **Sensitivity exposed as a public variable** — mouseSensitivity is public so testers and players can tweak it without changing code. This supports usability testing and stakeholder feedback (and is easy to put into an options menu later).
* **Pitch clamping to prevent flipping** — Clamping vertical rotation (Mathf.Clamp(xRotation, topClamp, bottomClamp)) avoids the camera flipping over when the player looks too far up/down. This is a basic UX safety measure and prevents disorienting behaviour which would invalidate tests of aiming and shooting.
* **Use of Quaternion.Euler and localRotation** — Applying a quaternion avoids gimbal-lock in Unity and localRotation is appropriate because the script is intended to live on the camera (so the camera rotates relative to the player object). This is standard practice for smooth, accurate camera control.
* **Cursor lock on Start** — Locking the cursor to the centre prevents the mouse from leaving the game window while testing, which is necessary for reliable input and automated / manual test runs.

**Simple numerical model (fast & reliable)** — The approach is lightweight (no physics, no unnecessary math) so it runs cheaply every frame. For an NEA project this keeps performance predictable on school machines and makes it easy to measure and tune.

I made the camera look separate from the movement so its own script can be changed or reused without touching the players movement to make debugging easier. I added clamps to prevent the camera flipping over itself and looking too far up/down, making gameplay smoother.

A screenshot of a computer program

AI-generated content may be incorrect.

PlayerMovement code (image 1/2)

A screen shot of a computer program

AI-generated content may be incorrect.

PlayerMovement code (image 2/2)

**2. Module: PlayerMovement.cs**

**Description**

PlayerMovement implements first-person movement using a CharacterController. It reads horizontal/vertical inputs, creates a world-space movement vector, applies gravity and jumping, checks grounding using a sphere check, and updates the controller each frame. It also tracks whether the player is moving.

PlayerMovement implements the first-person movement using a CharacterController. It reads movements and creates a movement vector, applies gravity and jumping, and using a ground check to determine whether the player is on the floor or not to disable jumping unless floored. I have chosen to do this straight after as it is another core requirement and is essential for any testing.

**Justification (why it’s in Stage 1)**

* **Core gameplay requirement:** Player navigation is essential for any playable level — enemies, reloading and UI can’t be tested without movement.
* **Safety & testing:** Ground check and gravity implement realistic jumping/falling behaviour which will be necessary for map testing (Stage 3).
* **Modular design:** Breaking movement out into its own class follows good decomposition practices and lets other systems (reload, enemy targeting) rely on a stable player interface.

**Key variables / Data dictionary**

| **Variable** | **Type** | **Purpose / Notes** | **Preset / Valid ranges** |
| --- | --- | --- | --- |
| speed | float | Player horizontal movement speed (units/sec). | Default 12f — measurable in tests. |
| gravity | float | Downward acceleration applied to player. | Default -9.81f (Earth-like). |
| jumpHeight | float | Target jump height in units. | Default 3f — can be tuned. |
| groundCheck | Transform | Position for ground-check sphere (usually at feet). | Must be set in Inspector. |
| groundDistance | float | Radius of ground-check sphere. | Default 0.4f — ensure suitable for character height. |
| groundMask | LayerMask | Layers considered as ground. | Set to terrain/geometry layers. |
| controller | CharacterController | Reference required component | assigned in Start() |
| velocity | Vector3 | Current vertical velocity (Y); used for gravity/jump. | N/A |
| isGrounded | bool | True if player is standing on ground. | N/A |
| lastPosition | Vector3 | For movement detection (isMoving). | N/A |

**Algorithm / Pseudocode (brief)**

On Start:

get CharacterController reference

Every Update:

isGrounded = Physics.CheckSphere(groundCheck.pos, groundDistance, groundMask)

if isGrounded and velocity.y < 0: velocity.y = -2f

x = Input.GetAxis("Horizontal")

z = Input.GetAxis("Vertical")

move = transform.right \* x + transform.forward \* z

controller.Move(move \* speed \* dt)

if Jump pressed and isGrounded:

velocity.y = sqrt(jumpHeight \* -2 \* gravity)

velocity.y += gravity \* dt

controller.Move(velocity \* dt)

isMoving = (transform.position != lastPosition) && isGrounded

lastPosition = transform.position

**Initial Test Plan (what to test for Stage 1)**

* **Movement tests**
  + Press W/S/A/D and verify player moves in intended directions relative to facing direction.
  + Check precise distance moved over one second roughly equals speed (distance ≈ speed × time).
* **Jump / Gravity tests**
  + Press Jump while grounded: player rises to approx jumpHeight then falls; verify velocity.y is set correctly.
  + Ensure player cannot jump when not grounded.
* **Ground detection**
  + Walk off a ledge — player should fall and isGrounded becomes false.
  + Place objects with different layer masks to ensure groundMask works.
* **Movement state**
  + Confirm isMoving toggles correctly when moving versus standing still (use logs or UI indicator).
* **Collision**
  + Ensure CharacterController colliders prevent passing through walls and remain above collider slope limits.

**Success Criteria (measurable)**

* Player moves at ~12 units/sec when holding forward (±10% tolerance).
* Player can jump and the max height is approximately 3 units.
* Ground-check reliably detects ground and prevents double-jumping.
* Character does not clip through simple colliders / floor at normal speeds.

**Known limitations / improvements**

* CheckSphere ground check radius and position must be tuned per model height — mention this in tests.
* Movement uses instantaneous acceleration; adding acceleration/decay will make movement feel smoother.
* Consider adding slope handling (max slope) and step offset settings on the CharacterController.
* Consider separating horizontal movement and vertical physics into clearer systems for maintainability.

**Justification — PlayerMovement (movement, gravity, jump)**

**What it does (brief):** uses a CharacterController to move the player based on WASD + camera forward/right, applies gravity and jumping via a vertical velocity, checks grounded state with Physics.CheckSphere, and detects movement state.

**Why it was coded this way**

* **Use of CharacterController (simplicity and control)**  
  The CharacterController handles collision and step-smoothing without needing to implement low-level Rigidbody physics. For a single-developer NEA this reduces bugs (no physics jitter or need to tune mass/drag) and makes movement deterministic and testable.
* **Camera-relative movement (transform.right / transform.forward)**  
  Movement uses the player/camera orientation so forward is always where the player is looking. That’s expected in FPS games and keeps input intuitive. It also matches the camera script, so player and camera coordinate correctly.
* **Gravity implemented via vertical velocity**  
  Adding gravity manually into a velocity.y variable gives precise control (and keeps horizontal movement separate from vertical physics). This makes jump height and fall behaviour measurable and tunable (important for success criteria like "jump ≈ 3 units").
* **Ground check using Physics.CheckSphere**  
  A sphere check near the feet is reliable for detecting ground on uneven terrain and prevents double-jumping. It’s simple and fast to run every frame and gives a clear boolean (isGrounded) for logic decisions, which improves robustness.
* **Resetting velocity.y when grounded**  
  Setting velocity.y = -2f (small negative) ensures the controller stays grounded and avoids jitter caused by tiny floating-point drift. Using a small negative value is a common Unity technique that is more stable than zero.
* **isMoving detection via lastPosition**  
  Tracking lastPosition to set isMoving enables simple player-state checks (useful for footstep sounds, animations, and unit tests). It’s cheap and avoids noisy input checks.
* **Public, tunable fields**  
  speed, jumpHeight, gravity, and ground-check settings are public so you can tune them during playtesting without recompiling. That supports iterative improvement based on stakeholder feedback.

**Combined Stage 1 — Integration & Rationale**

**Why these two together?**

* Camera and movement are dependent: movement relative to camera orientation provides the FPS feel. Implementing both ensures the core loop (move + look) is testable immediately and lets you integrate shooting/reloading in Stage 2 without needing rework.
* They are discrete modules (decomposition) so you can iterate on each quickly: e.g., change sensitivity without touching movement code.

**Stage 1 Acceptance Tests (end-of-stage checklist)**

1. Player can look smoothly in all directions; pitch is clamped to [-75°, 75°].
2. Player can move in all directions using WASD relative to camera.
3. Player can jump only when grounded and gravity brings player back down.
4. Movement speed and jump height roughly match specified values.
5. Cursor locks on play start.
6. No null reference exceptions when scripts run (CharacterController and groundCheck assigned).

Record results (pass/fail) for each test and any deviations — these are your Stage 1 testing artifacts to include in the NEA.

**How this aligns with OCR marking criteria**

* **Decomposition & Design:** The code modules show clear decomposition (camera vs movement). Documenting variables and algorithms demonstrates planned structure and data use.
* **Algorithmic understanding:** Pseudocode and test plans show you understand the algorithms and edge cases (ground check, gravity integration, clamping).
* **Measurable success criteria:** I provided numerical success measures (speed, jump height, clamp angles) so tests are objective.
* **Testing evidence:** The test steps and acceptance checklist are directly usable for your stage testing documentation (screenshots/video + pass/fail table).

**How these choices help meet OCR criteria / project goals**

* **Decomposition & maintainability:** Camera and movement are separate modules — shows good software design and makes future features (e.g., head-bobbing, sprinting) easier to add without breaking core functionality.
* **Testability / Measurability:** Gravity, jumpHeight, speed and clamp values are explicit parameters. That makes it straightforward to create measurable success criteria and automated/manual tests (e.g., “player jumps to ~3 units”).
* **Performance & robustness:** Both modules are lightweight, deterministic, and avoid expensive physics calculations. This reduces the chance of performance issues on school hardware and improves reproducibility during marking.
* **Player experience:** These implementations reflect standard FPS behaviour, so testers will find controls intuitive. That reduces false negatives in usability testing (i.e., testers complaining about controls rather than the reload mechanic you’re testing).
* **Extensibility:** Because movement and camera are isolated, you can later attach animations, add strafing friction, smoothing, crouch, or swap to Rigidbody-based physics if you want more advanced behaviour — without rewriting the other systems.

**Optional improvements (explainable if a marker asks)**

If you need to defend why these haven’t been done yet, you can say:

* **Yaw applied to camera only** — In some FPS setups yaw is applied to the player GameObject (rotating the whole body) and pitch to the camera only. Right now the script rotates its transform; that is fine if the script is on the camera. If you later need root yaw (for moving character model or hit detection), you can easily change yaw to rotate the parent (player) and leave pitch on camera — separation already enables that change.
* **Smoothing & input filtering** — Not added yet to keep first-stage testing simple. You can justify delaying this until Stage 8 (polishing) because it doesn’t affect correctness.
* **CharacterController vs Rigidbody** — Chosen for deterministic, simple collisions and easier control in small project scope. A Rigidbody approach would be more realistic but requires more tuning (mass, forces, interpolation) — an unnecessary risk early in development.

Screenshot of character created