A2 “shadowing”—Project Documentation

Gameplay Mechanics and Test Design

June 2026

Contents

1 Project Overview **3**

1.1 Project Name . . . 3

1.2 Project Type . . . 3

1.3 Development Environment . . . 3

1.4 Project Assets . . . 4

2 Game Mechanics and Gameplay **4**

2.1 Core Gameplay Loop . . . 4

2.2 Player Controls . . . 5

2.3 Click-to-Move Mechanism . . . 6

2.4 Shadow Replay Mechanism . . . 6

2.4.1 Recording Phase . . . 7

2.4.2 Release Phase . . . 7

2.4.3 Playback Phase . . . 7

2.5 Enemy Target-Acquisition Mechanism . . . 7

2.6 Enemy Types and Behavior . . . 8

2.6.1 Scout . . . 8

2.6.2 Archer . . . 9

2.6.3 Heavily Armored Gate Guard . . . 9

2.6.4 Scout–Guard Coordination . . . 10

2.7 Camera Mechanism . . . 10

3 Project Structure **11**

3.1 Core Script Structure . . . 11

3.2 Scene Files . . . 12

4 Game Testing **12**

4.1 Test Objectives . . . 12

4.2 Test Environment . . . 12

4.3 Automated Test Overview . . . 13

4.4 Test Utility Class . . . 13

4.5 Camera Tests . . . 14

4.5.1 Following a Moving Target . . . 14

4.5.2 World-Boundary Clamping . . . 14

4.5.3 Scene Camera Configuration . . . 14

4.6 Enemy AI Tests . . . 15

4.6.1 Visibility Requirements . . . 15

4.6.2 Shadow-Priority Targeting . . . 15
4.6.3 Archer Spawns Homing Arrow . . . 15
4.6.4 Scout Raises an Alarm . . . 15
4.7 Map Navigation Tests . . . 16
4.7.1 NavMesh Paths in the Main Scene . . . 16
4.7.2 Runtime NavMesh Movement . . . 16
4.7.3 Runtime NavMesh Boundary Validation . . . 16
4.8 Manual Test Cases . . . 16
4.9 Test Coverage and Gaps . . . 17
4.10 Running the Tests . . . 18
4.10.1 Running in Unity Editor . . . 18
4.10.2 Running from the Command Line . . . 18

5 Conclusion **18**

1 Project Overview

1.1 Project Name

A2 “shadowing”

1.2 Project Type

This project is a 3D top-down stealth-puzzle game prototype developed in Unity 6. Players right-click the ground to move their character. Using the “shadow replay” mechanic, they create a shadow duplicate that retraces their past movements, distracting enemies, changing their facing directions, luring guards away, and ultimately allowing the player to pass through areas under enemy control.



Figure 1: Main gameplay view

1.3 Development Environment

Table 1: Project development environment

Item	Details
Game engine	Unity 6000.3.12f1
Programming language	C#
Render pipeline	Universal Render Pipeline 17.3.0
Input system	Unity Input System 1.19.0
Navigation system	Unity AI Navigation 2.0.11 / NavMesh
Camera	Custom MOBA-style follow camera; Cinemachine is also included as a project dependency
Testing framework	Unity Test Framework 1.6.0 / NUnit

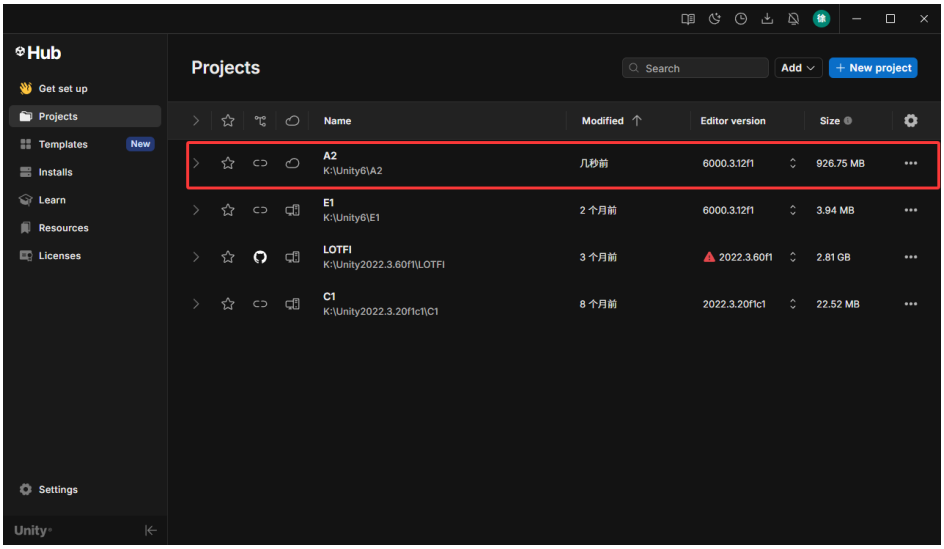


Figure 2: Unity project version and settings

1.4 Project Assets

Table 2: Project asset usage

Asset	Purpose
ToonyTinyPeople / TT_RTS	Player and enemy character models, including archers, scouts, and heavily armored guards.
SimpleFantasy	Prefabs for arrows, weapons, and related objects.
Suntail Village	Environmental assets such as village buildings, scenery, and vegetation.
FunkyCode	Resources related to 2D lighting and shadows.

2 Game Mechanics and Gameplay

2.1 Core Gameplay Loop

The game is not centered on direct combat. Instead, the player uses their "past self" to manipulate enemy decisions. The player observes enemies, plans a route, records a shadow trajectory, then releases the shadow to distract enemies and create an opportunity to pass.

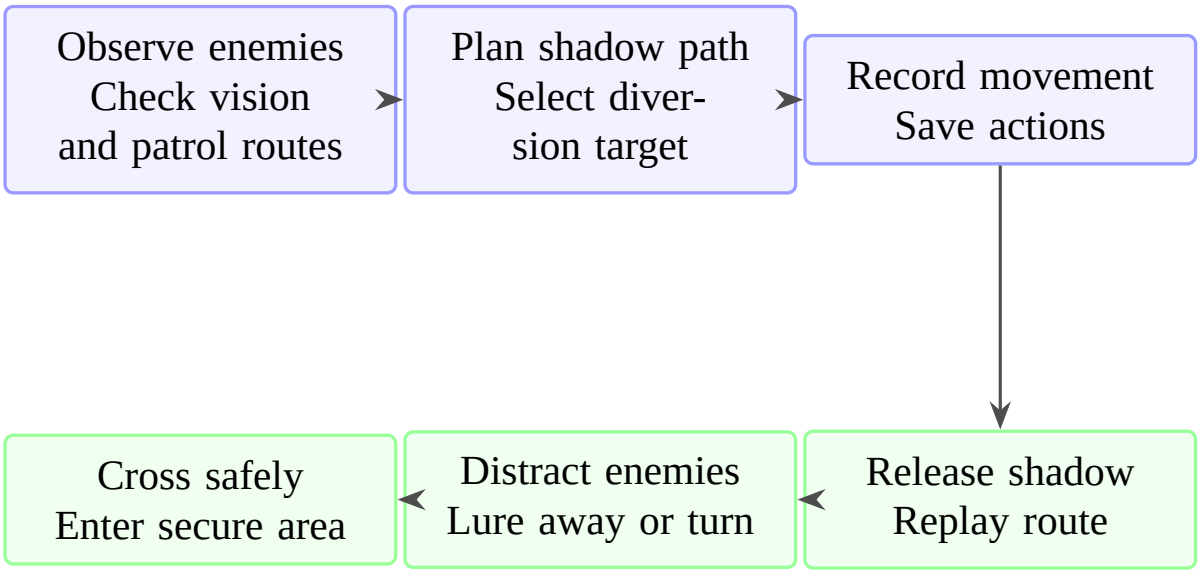


Figure 3: Core gameplay loop

The core steps are:

1. Observe enemy positions, facing directions, and patrol/guard behavior.
2. Plan a shadow path that will attract an enemy.
3. Press the recording key, move the player, and capture the trajectory.
4. Release the shadow duplicate after recording.
5. The shadow follows the recorded route, and enemies prioritize detecting and tracking it.
6. Pass through the dangerous area when enemies leave their posts or turn away.

2.2 Player Controls

The controls are documented with the following table and keyboard diagram; no additional screenshots are required.

Table 3: Player controls

Action	Default input	Description
Move	Right-click ground	NavMeshAgent automatically finds a route to the clicked destination.
Record shadow	R key	Starts recording player position and rotation, up to 10 seconds by default.
Release shadow	F key	Spawns a shadow duplicate that follows the recorded trajectory.
Zoom camera	Mouse wheel	Adjusts top-down camera distance.
Pan camera	WASD / cursor near screen edge	Moves the camera view in MOBA/RTS style.



Figure 4: Keyboard and mouse control guide

2.3 Click-to-Move Mechanism

Player movement is implemented in `MoveToClickPoint.cs`, with the following main logic:

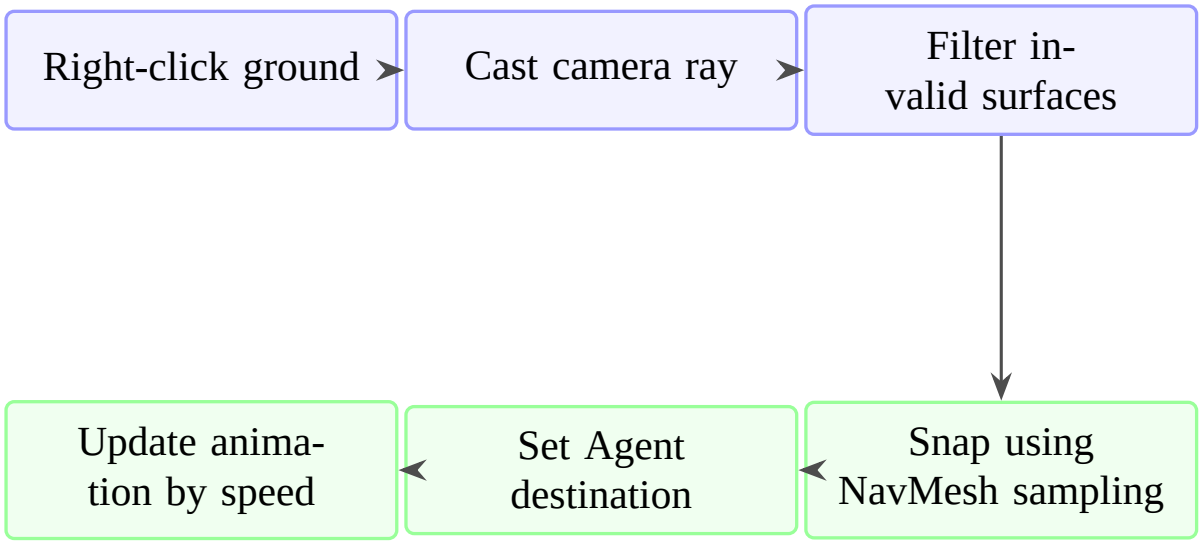


Figure 5: Click-to-move processing flow

This control scheme resembles MOBA and RTS games and is suitable for top-down stealth levels.



Figure 6: Ground indicator for click-to-move

2.4 Shadow Replay Mechanism

The shadow system is the central mechanic of the project and consists of three principal scripts:

Table 4: Responsibilities of shadow-system scripts

Script	Responsibility
`ShadowReplayFrame.cs`	Stores a trajectory sample containing timestamp, position, and rotation.
`ShadowReplayRecorder.cs`	Records the player trajectory and handles shadow release.
`ShadowReplayPlayback.cs`	Replays the recorded samples using interpolation.

2.4.1 Recording Phase

Pressing R starts recording. At fixed sampling intervals, the system records the player's position and orientation; the default recording duration is 10 seconds. Samples are stored as a collection of `ShadowReplayFrame` objects.

Key recording rules:

- Whether a second recording command restarts the process is configurable.
- Each frame contains a timestamp, position, and rotation.
- Recording stops automatically at the maximum duration.
- At least two valid frames are required to release a shadow.

2.4.2 Release Phase

Pressing F spawns the shadow prefab at the position of the first recorded frame. The system then configures the new shadow at runtime:

- Adds or preserves the `ShadowReplayPlayback` component.
- Adds `EnemyTarget` and sets its type to `Shadow`.
- Disables `NavMeshAgent`, `NavMeshObstacle`, colliders, and rigidbody interactions on the shadow.
- Optionally disables ordinary scripts to prevent the shadow from responding to player input.
- If configured to replace the previous shadow, releasing a new one destroys the old one.

2.4.3 Playback Phase

Playback is handled by `ShadowReplayPlayback.cs`. The shadow interpolates between adjacent recorded frames according to the current playback time:

- Position transitions are smoothed with `Vector3.Lerp`.
- Rotation transitions are smoothed with `Quaternion.Slerp`.
- Animation switches among Idle / Walk / Run based on movement speed.
- After playback, the shadow can be destroyed automatically or remain at its final position.

The shadow neither attacks enemies nor physically blocks enemies or the player. Its purpose is to be seen and thereby induce incorrect enemy actions.

For a demonstration, see the accompanying video "影子.mp4" (Shadow.mp4).

2.5 Enemy Target-Acquisition Mechanism

Enemies use `EnemyTarget.cs` for unified target recognition. Both the player and the shadow are registered as potential targets.

Table 5: Enemy target types

Type	Meaning
Player	The actual player character.
Shadow	The shadow duplicate.

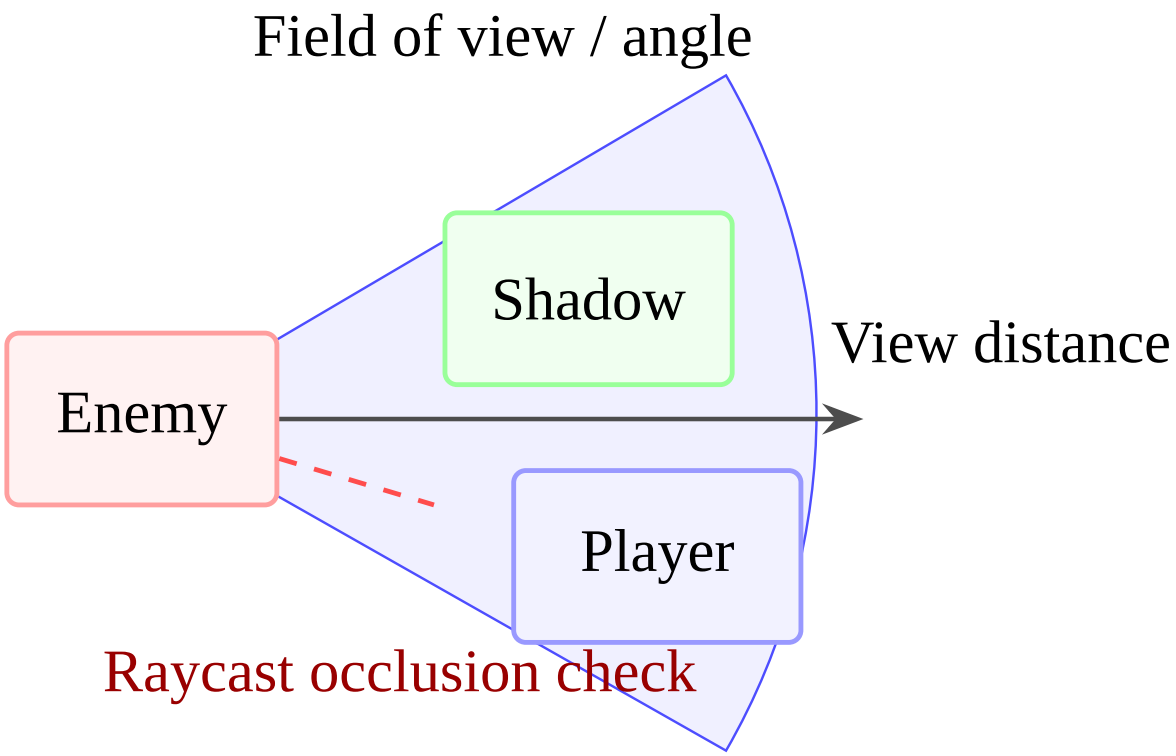


Figure 7: Enemy fan-shaped field of view

Enemy visibility checks comprise three steps: distance, field-of-view angle, and occlusion. In most enemy behaviors, `preferShadow` is enabled, meaning an enemy prioritizes the shadow as its target. This allows the shadow to function as a decoy.

For a demonstration, see the accompanying video “演示.mp4”(Demo.mp4).

2.6 Enemy Types and Behavior

2.6.1 Scout

The scout is implemented by `ScoutAlarmAI.cs`; its main role is to detect the player and raise an alarm.

Table 6: Scout states

State	Description
Watching	Watches for targets from its position.
ChasingShadow	Pursues a detected shadow.
RunningToAlarm	Runs to an alarm point after confirming player detection.
Returning	Returns to its initial position after losing the shadow.
AlarmRaised	Raises the alarm upon reaching the alarm point.

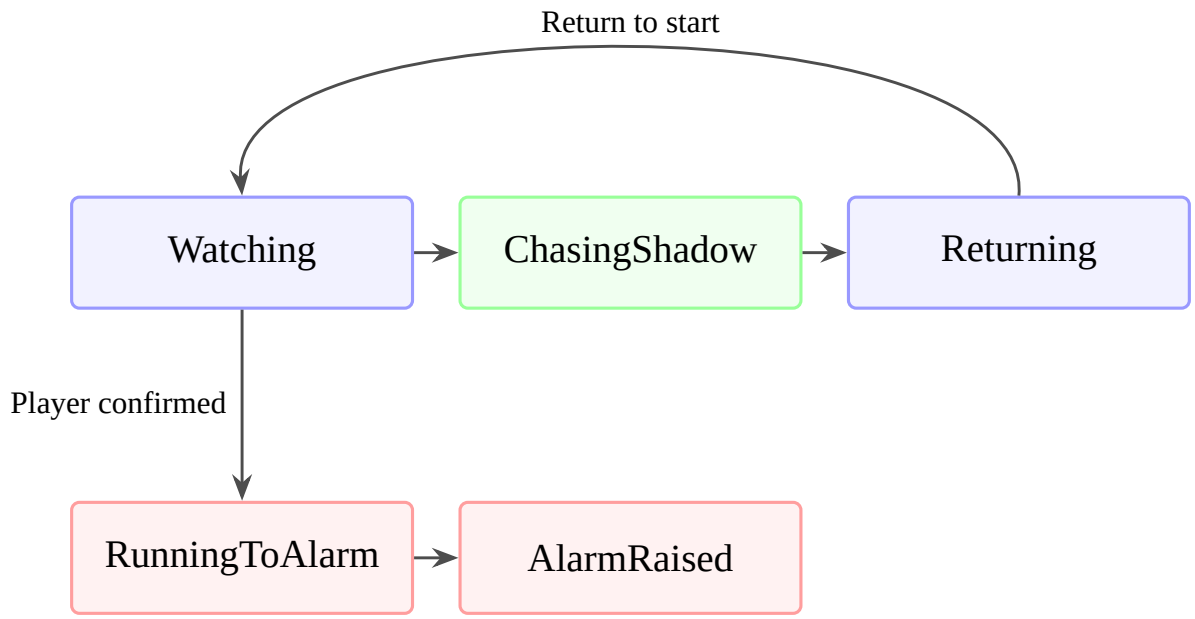


Figure 8: Scout state machine

2.6.2 Archer

The archer is implemented by `ArcherSentinelAI.cs` and `HomingArrowProjectile.cs`. It remains stationary and controls a fixed area.

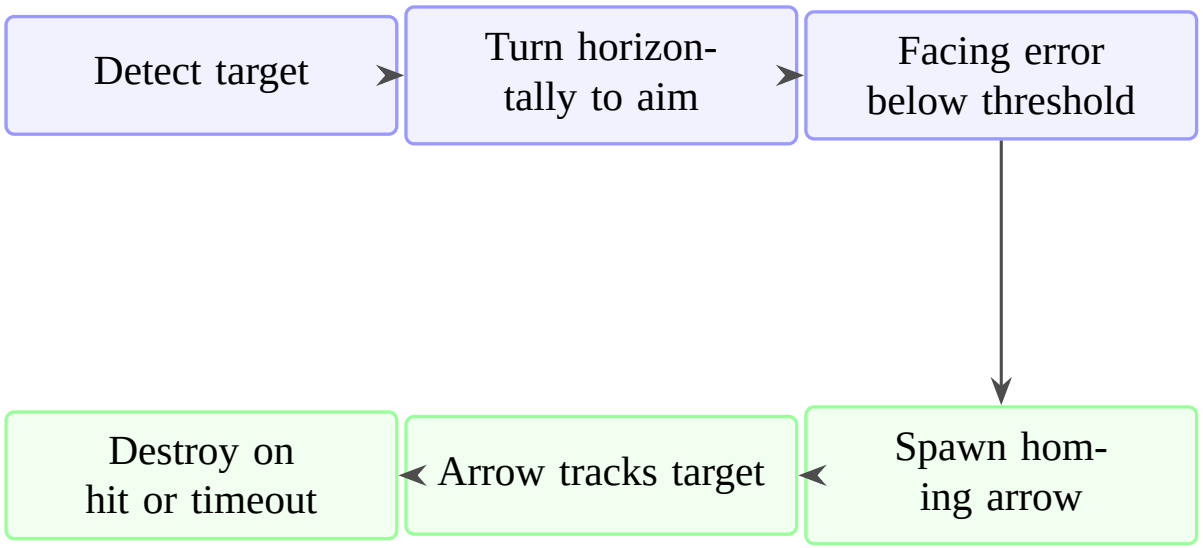


Figure 9: Archer behavior flow

The intended interaction is not to lure the stationary archer away, but to force it to turn so the player can pass through a blind spot on the other side.

2.6.3 Heavily Armored Gate Guard

The heavily armored gate guard is implemented by `HeavyGateGuardAI.cs`. It protects doorways, bridges, and narrow passages.

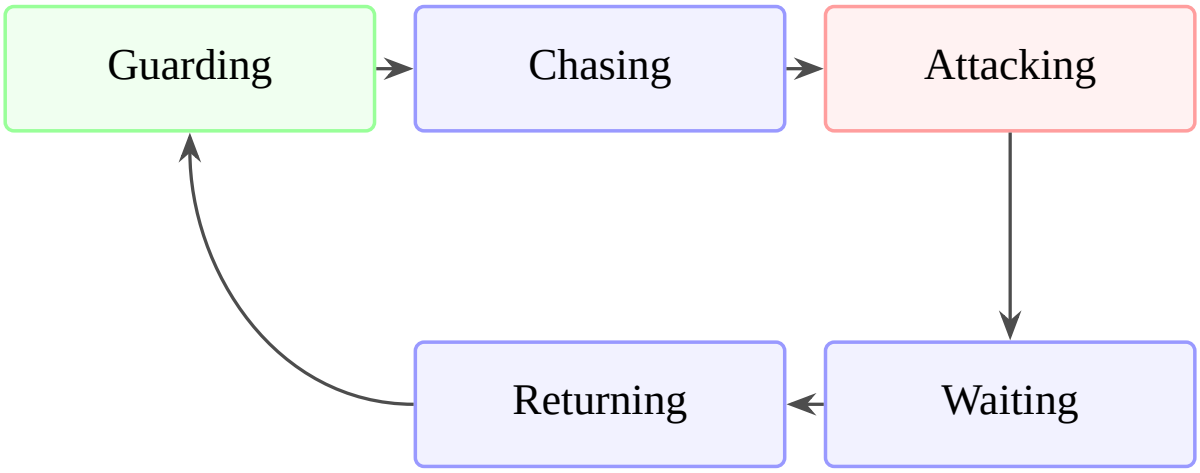


Figure 10: Heavily armored guard state machine

The guard has a maximum permitted distance from its post and cannot be lured away indefinitely. The player must plan a sufficiently enticing shadow route so the guard leaves the doorway briefly, providing an opening.

2.6.4 Scout–Guard Coordination

`ScoutAlarmGuardRedirector.cs` links scout behavior with that of a designated heavily armored guard. Once the scout confirms that it has detected the player, the guard's post can be redirected to the player's exposed location, increasing level difficulty.

For attack behavior, see the accompanying video "攻击方式.mp4"(Attack Methods.mp4).

2.7 Camera Mechanism

`CameraFollow.cs` implements a MOBA/RTS-style top-down camera featuring target following, velocity look-ahead, mouse-look offset, zoom, panning, world-boundary clamping, and occlusion-cone transparency.

Table 7: Camera features

Feature	Description
Target following	Follows the player, with optional smooth movement.
Velocity look-ahead	Shifts the view slightly in the player's movement direction.
Mouse-look offset	Offsets the view as the cursor approaches a screen edge.
Zoom	Mouse wheel controls camera distance and pitch.
Panning	Supports WASD and edge-of-screen panning.
Boundary clamping	Clamps the camera focal point to configured world bounds.
Occlusion cone	Makes buildings and trees semitransparent when they obscure the player.



Figure 11: Top-down follow-camera view

3 Project Structure

3.1 Core Script Structure

The script-directory layout is shown directly in monospaced text, without a screenshot.

```
Assets/Scripts/
|-- ShadowReplayFrame.cs
|-- ShadowReplayRecorder.cs
|-- ShadowReplayPlayback.cs
|-- Camera/
|   `-- CameraFollow.cs
|-- Click/
|   |-- ClickGroundIndicatorRuntime.cs
|   `-- ClickMoveIndicator.cs
|-- Cursor/
|   `-- GameCursor.cs
|-- Enemies/
|   |-- ArcherSentinelAI.cs
|   |-- EnemyAnimatorUtility.cs
|   |-- EnemyTarget.cs
|   |-- HeavyGateGuardAI.cs
|   |-- HomingArrowProjectile.cs
|   |-- ScoutAlarmAI.cs
|   `-- ScoutAlarmGuardRedirector.cs
|-- Movement/
|   `-- MoveToClickPoint.cs
`-- Tests_PlayMode/
    |-- CameraFollow_Test.cs
    |-- EnemyAI_PlayModeTests.cs
    |-- MapNavigation_PlayModeTests.cs
    `-- PlayModeTestUtilities.cs
```

3.2 Scene Files

Table 8: Scene files

Scene	Path	Description
GamePlay	`Assets/Scenes/GamePlay.unity`	Main gameplay scene.
Demo_Environment	`Assets/Scenes/Demo_Environment.unity`	Scene showcasing the environment and assets.

4 Game Testing

4.1 Test Objectives

Testing aims to determine whether core game systems are stable and behave as intended. Focus areas include click-to-move, shadow recording and replay, enemy detection of the player and shadow, shadow-priority targeting, enemy state machines, camera following and world bounds, NavMesh navigation, and automated-test coverage.

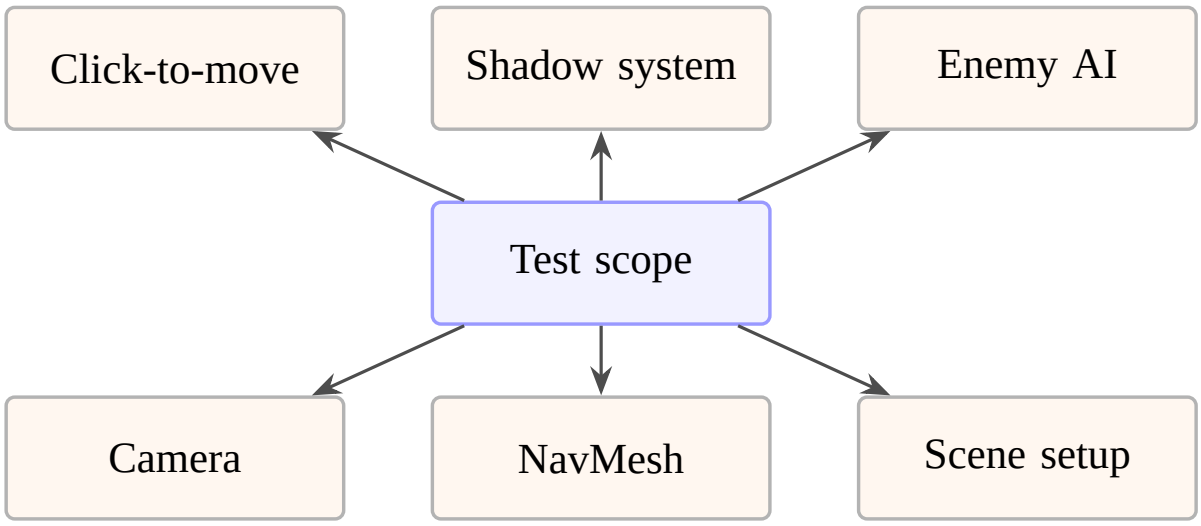


Figure 12: Test objectives and system scope

4.2 Test Environment

Table 9: Test environment

Item	Details
Operating system	Windows 10
Unity version	6000.3.12f1
Test framework	Unity Test Framework 1.6.0
Test types	Play Mode tests + some standard NUnit tests
Test directory	`Assets/Scripts/Tests_PlayMode/`
Test assembly	`Tests_PlayMode`

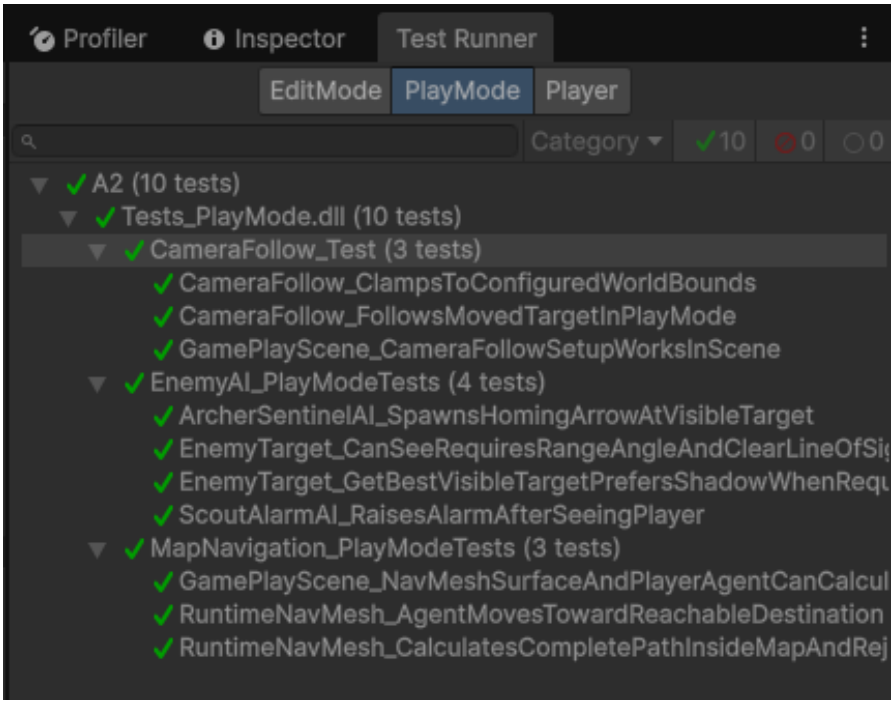


Figure 13: Unity Test Runner results

4.3 Automated Test Overview

Table 10: Automated test distribution

Test file	Count	Coverage
`CameraFollow_Test.cs`	3	Camera follow, bounds, scene setup.
`EnemyAI_PlayModeTests.cs`	4	Vision detection, targeting priority, archer, scout.
`MapNavigation_PlayModeTests.cs`	3	GamePlay scene NavMesh, runtime NavMesh, path computation.
Total	10	Core camera, enemy AI, and navigation logic.

4.4 Test Utility Class

`PlayModeTestUtilities.cs` provides shared test helpers.

Table 11: PlayModeTestUtilities helpers

Method/class	Purpose
`WaitForFrames(int)`	Waits for a specified number of frames; useful for behaviors that require several frames to take effect.
`WaitUntil(Func<bool>, float, string)`	Waits for a condition until timeout; reports a clear error upon failure.
`SetPrivateField<T>()`	Sets private component fields via reflection to test varying parameters.
`GetPrivateField<T>()`	Reads private component fields via reflection.
`CreateEnemyTarget()`	Creates a test object with an EnemyTarget component.
`CreateGameObject()`	Creates a test GameObject quickly.
`RuntimeNavMesh`	Builds a planar NavMesh at runtime for scene-independent navigation tests.

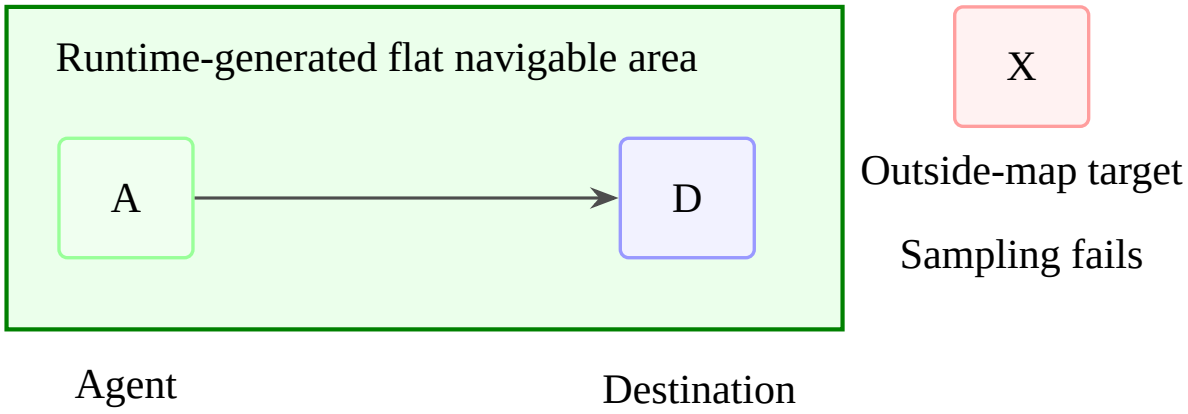


Figure 14: RuntimeNavMesh test scene

4.5 Camera Tests

4.5.1 Following a Moving Target

Objective: Verify that the camera correctly follows a target after it moves.

Procedure: Create a runtime target and a test camera; disable smoothing, zoom, look-ahead, and other dynamic features. After waiting several frames, verify camera position and rotation. Move the target and repeat the checks.

Expected result: Camera position error does not exceed 0.02 units; rotation error does not exceed 0.02 degrees; field of view matches the configured value.

4.5.2 World-Boundary Clamping

Objective: Verify that the camera world-boundary constraints work.

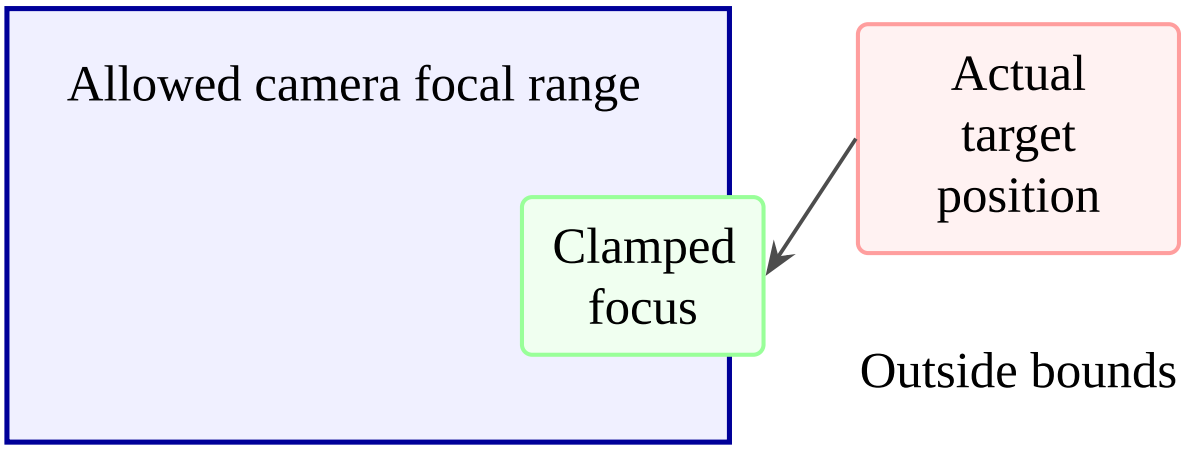


Figure 15: Camera world-boundary clamping

Expected result: Even if the target moves outside the boundary, the camera focal point stays within the configured limits.

4.5.3 Scene Camera Configuration

Objective: Verify that the camera in the main scene has a complete configuration.

The test loads the GamePlay scene, finds CameraFollow, checks the main camera and target object, verifies MoveToClickPoint and NavMeshAgent, then moves the player target and waits for the camera to follow again.

4.6 Enemy AI Tests

4.6.1 Visibility Requirements

Objective: Verify that enemy visibility requires all three conditions: distance, angle, and an unobstructed line of sight.

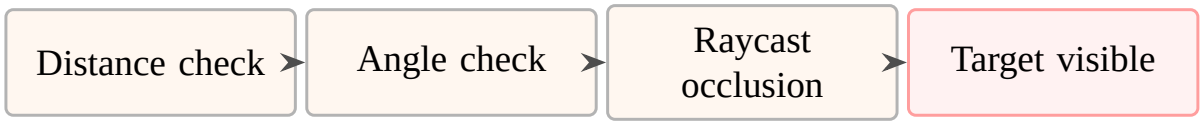


Figure 16: Three visibility conditions

Expected result: A target directly ahead is visible; a target to the side is not; an occluded target is not visible.

4.6.2 Shadow-Priority Targeting

Objective: Verify that an enemy prioritizes the shadow target when the shadow-priority option is enabled.

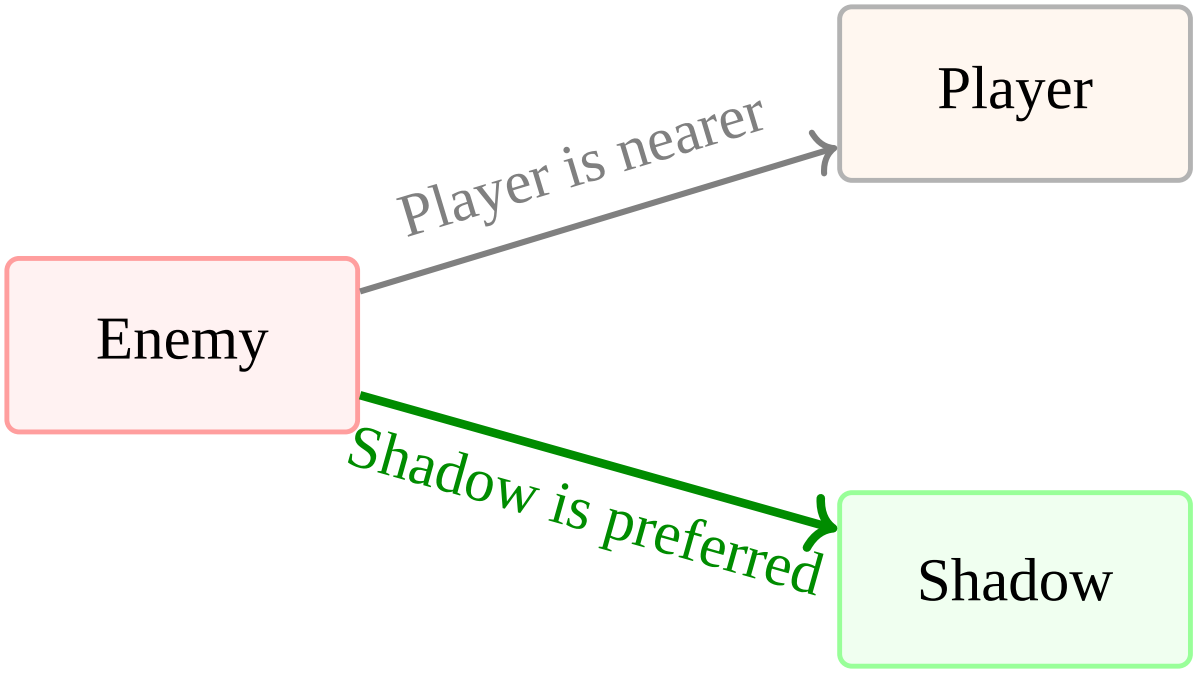


Figure 17: Priority between player and shadow targets

Expected result: With `preferShadow` enabled, the method returns the shadow even when it is farther away; when disabled, it returns the closer player.

4.6.3 Archer Spawns Homing Arrow

Objective: Verify that an archer spawns a homing arrow after detecting a visible target.

Procedure: Create a player target, a test arrow prefab, and an archer object. Add `ArcherSentinelAI`, set vision and firing parameters using reflection, and wait for a `HomingArrowProjectile` to appear in the scene.

4.6.4 Scout Raises an Alarm

Objective: Verify that a scout raises an alarm after continuously observing the player.

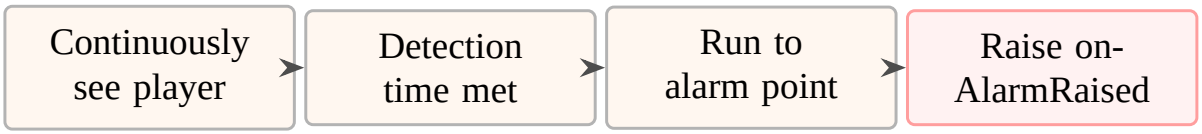


Figure 18: Scout alarm sequence

Expected result: Upon seeing the player, the scout enters the alarm sequence. The `onAlarmRaised` event is invoked, and the scout stops moving once the alarm is raised.

4.7 Map Navigation Tests

4.7.1 NavMesh Paths in the Main Scene

Objective: Verify that the GamePlay scene contains a usable NavMesh and the player can compute a route.

Expected result: A valid NavMesh exists in GamePlay; the player Agent can bind to it; at least one complete path is available near the player.

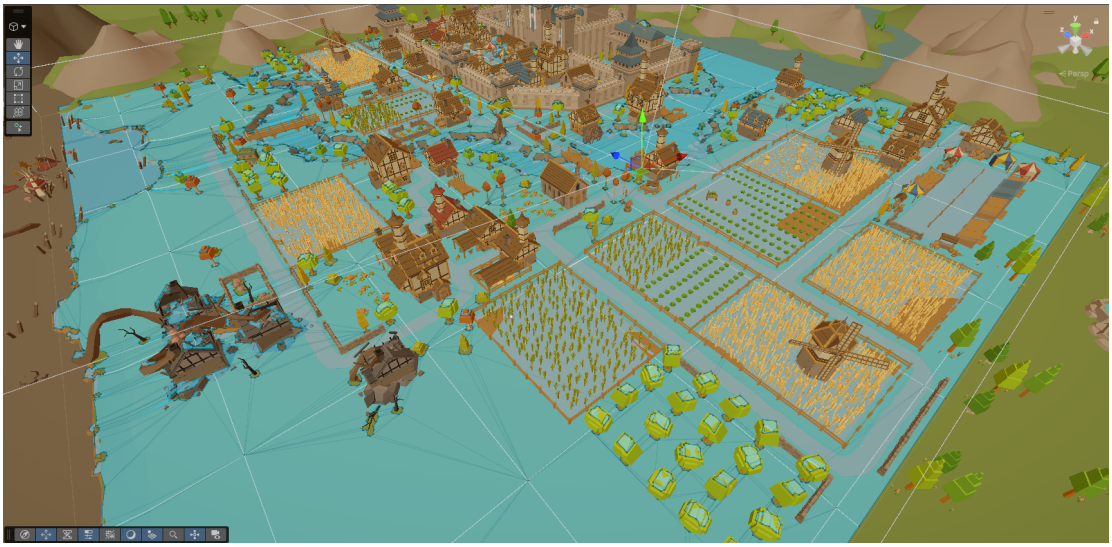


Figure 19: Baked NavMesh in the GamePlay scene

4.7.2 Runtime NavMesh Movement

Objective: Verify that a runtime-generated NavMesh supports actual Agent movement.

Expected result: The Agent can be placed on the runtime NavMesh, accepts a destination, reports `PathComplete`, and moves toward the target.

4.7.3 Runtime NavMesh Boundary Validation

Objective: Verify that the runtime NavMesh distinguishes destinations inside and outside the navigable map.

Expected result: Path computation succeeds for in-map destinations; outside-map targets cannot be sampled onto the NavMesh.

4.8 Manual Test Cases

Table 12: Manual test cases

Test case	Test	Procedure	Expected result
TC-MOVE-001	Right-click movement	In GamePlay, right-click walkable ground.	Player reaches clicked point; animation changes correctly.
TC-MOVE-002	Invalid-click filtering	Right-click a wall, obstacle, or inaccessible area.	Player does not move, or destination is rejected.
TC-SHADOW-001	Shadow recording	Press R and move for 10 seconds.	Trajectory is recorded; recording stops on time.
TC-SHADOW-002	Shadow release	After recording, press F.	Shadow spawns at starting point and follows recorded route.
TC-SHADOW-003	Shadow collision	Release shadow and let it pass near enemies or obstacles.	Shadow does not push objects or participate in NavMesh navigation.
TC-ENEMY-001	Enemy detects player	Player enters enemy field of view.	Enemy detects player and enters tracking/alarm state.
TC-ENEMY-002	Shadow priority	Player and shadow both enter enemy field of view.	Enemy selects shadow first.
TC-ARCHER-001	Archer fires	Move shadow into archer field of view.	Archer turns and fires a homing arrow.
TC-SCOUT-001	Scout alarm	Keep player in scout field of view.	Scout runs to alarm point and raises alarm.
TC-GUARD-001	Gate guard leaves post	Release shadow to lure gate guard away.	Guard follows shadow, then returns to its post.
TC-CAMERA-001	Camera zoom	Scroll mouse wheel.	Camera distance and angle change smoothly.
TC-CAMERA-002	Camera occlusion	Place building between player and camera.	Obstruction becomes semitransparent; player remains visible.
TC-LEVEL-001	Complete level	Complete all areas from the starting point.	Player reaches endpoint using the shadow mechanic.

4.9 Test Coverage and Gaps

Table 13: Test coverage matrix

Module	Current coverage	Notes
Camera	Covered	Follow, world bounds, and scene configuration.
Enemy AI	Partially covered	Vision, shadow priority, archer firing, scout alarm.
Map navigation	Covered	GamePlay and runtime NavMesh, pathfinding and Agent movement.
Movement system	Not automated	Mouse input and animation switching tests are still needed.
Shadow system	Not automated	Recording, release, and interpolation tests are needed.
Heavy-guard AI	Not automated	Full state-machine tests are needed.
Scout–guard coordination	Not automated	Guard-post redirection tests are needed.

The following additional tests are recommended:

1. After a 10-second recording, verify frame count and first/last timestamps.

2. After releasing the shadow, verify that `EnemyTargetKind` is `Shadow`.
3. Supply two recorded frames and verify position interpolation at an intermediate time.
4. Test all heavy-guard transitions among `Guarding`, `Chasing`, `Waiting`, and `Returning`.
5. Verify that confirming player detection allows the scout to redirect the gate guard’s post.

4.10 Running the Tests

4.10.1 Running in Unity Editor

1. Open the Unity project.
2. Navigate to `Window > General > Test Runner`.
3. Switch to `PlayMode`.
4. Click `Run All` to run every Play Mode test.

4.10.2 Running from the Command Line

The following example uses PowerShell.

```
& "E:\Unity\Editor\Unity.exe" -batchmode `
  -projectPath "<ProjectRoot>" `
  -runTests -testPlatform PlayMode `
  -testResults "<OutputDirectory>\test-results.xml"
```

Note: The project must not already be open in another Unity instance. Otherwise, Unity reports “Multiple Unity instances cannot open the same project”.

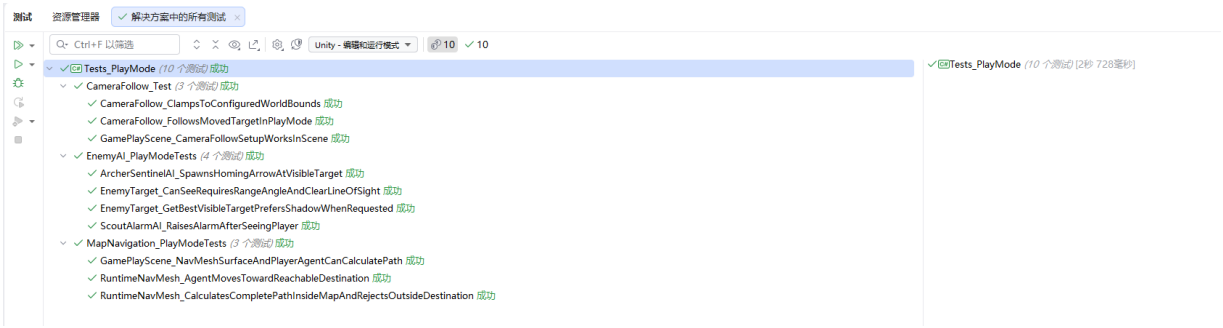


Figure 20: Test results from an external test runner (original software UI)

5 Conclusion

This Unity gameplay prototype combines stealth and puzzle solving through a shadow-replay mechanic. The player creates a duplicate from the previous 10 seconds of movement, using it to distract enemies, influence their behavior, and open a passage. The prototype includes click-to-move, shadow recording and replay, unified enemy targeting, scout alarms, homing arrows fired by archers, heavily armored gate guards, a MOBA-style camera, and NavMesh navigation.

The project has established Play Mode automated tests for the camera, enemy AI, and navigation systems. Future development should prioritize automated tests of shadow recording and replay, the heavy-guard state machine, and the complete level progression.