Assignment 5: Spy Camera — Teacher Guide

At a glance

Learning objective	First scoring! Students aim the satellite at the opponent every second and take photographs ONLY when an `if … then …` guard says the shot will count: the every-second replacement for "wait until ready, then shoot."
Time estimate	60–90 minutes
Project setup	New project on the **AsteroidBee 2026** game, **Graphical Editor**
Prerequisite assignment	Assignments 1–4
Blocks introduced	From the AsteroidBee categories: `get att to other to [array]`, `set AttitudeTarget` (a dropdown choice on the `set PositionTarget ▾ [array]` block), `is facing other`, `is camera on`, `take pic`, `get <my/other> ▾ energy` (dropdown set to **my**), and (as a mid-class refinement) `check in light <me/other> ▾` (dropdown set to **other**). The array declare block (`type: ▾ name: __ length: __ …`) is reused from earlier assignments.

The one idea this assignment teaches

Every student will want to write the same plan: turn toward the enemy, wait until we're facing them, then take the picture. It is a perfectly good plan, and there is no "wait" block anywhere in the editor. There can't be: the main page re-runs from the top every second, so a program can never sit inside one block waiting for something to finish. The replacement for waiting is **asking**. Each second the program asks, "Am I facing them right now? Is the camera ready right now?", and only on the seconds when every answer is yes does it shoot. The `if … then …` block, re-asked every second, **is** the wait.

The aiming half of the program teaches the same idea from the other direction. `get att to other to [aim]` fills a 3-slot array with the direction that points at the opponent, measured fresh that second. Feeding it into `set AttitudeTarget` tells the satellite "point that way." Run once, that's a single glance. Re-run every second, the pair becomes a **camera gimbal**: the swiveling mount that keeps a movie camera locked on its subject. Nothing in the blocks says "track the target"; tracking emerges from re-running the same two blocks 180 times. (In this assignment the opponent sits still, but the same blocks would track a moving one. Worth saying out loud.)

There's also a budget lesson hiding in the game rules: every `take pic` attempt costs 1.0 energy even when the photo fails, and energy only recharges (+0.5 per second) in the Light Zone. So checking before shooting isn't just tidier; it is the difference between a satellite that scores all match and one that goes dark.

> ⏱ **Every second:** the main page runs again from the top. `get att to other to [aim]` re-measures where the opponent is, `set AttitudeTarget` re-points the satellite, and the `if … then …` block re-asks its questions. The program never waits for anything; it just gets asked again one second later, and that is the waiting.

Before class

- [ ] Re-read the **Photographs**, **Energy**, and **Light and Dark Zones** sections of game-rules.md. The numbers in them (3-second cooldown, 1.0 energy per attempt, +0.5 per second recharge, swap at t = 60 and t = 150) drive the whole lesson.
- [ ] Build the reference solution yourself once (see reference-solution.md) and simulate a full 180 seconds, so you have seen the score climb, stall, and resume.
- [ ] Have the arena diagram ready to project or print:
- [ ] Know where the **scoreboard** appears in the simulation viewer; students will be watching it like a stock ticker.

- [ ] Confirm every student can create a new **Graphical Editor** project on the **AsteroidBee 2026** game and still remembers the two pages (init runs once, main re-runs every second) from Assignments 1–4.

Walkthrough: what to say and do

1. **Brief the mission from the game rules.** Say: "Today you finally score points: by photographing the rival satellite." Put up the rules for a successful photo: the camera must be on (it shuts off for **3 seconds** after every attempt), you must be **facing the opponent**, the opponent must be **in the light**, and you must be **at least half a meter away**. Then the kicker: **every attempt costs 1.0 energy, even a failed one**, and energy recharges only +0.5 per second, and only in the Light Zone. Ask the class: "So what could go wrong if we just shoot every chance we get?"

2. **Introduce the aiming idea.** Draw it on the board: the opponent is at some direction from us, and that direction is re-measured every second. The block `get att to other to [array]` (AsteroidBee Pictures category) fills a 3-slot array with the direction pointing at the opponent. Feed that array into the movement block `set PositionTarget ▾ [array]` with its **dropdown switched to** `set AttitudeTarget`: "attitude" is spacecraft-speak for which way you're pointing. Say: "Re-run every second, these two blocks are a camera gimbal: the swivel mount that keeps a camera locked on its target."

3. **Set up the init page.** On the **init** page, drag an array declare block (`type: ▾ name: __ length: __ …`) into the **"global variables"** slot. Type: float, name **aim**, length **3**. Ask the class why it goes here and not on the main page (declares only live in the global-variables slot, and a variable declared here is remembered between seconds).

4. **Build version 1 on the main page.** Into the **"loop"** slot, in this order:

5. `get att to other to [aim]`: the array block comes pre-attached; pick **aim** from its dropdown.
6. `set PositionTarget ▾`, dropdown switched to `set AttitudeTarget`, with the whole-array **aim** block plugged into its socket.
7. An `if … then …` block. For its condition, use one `[a] and ▾ [b]` block from Logic, and plug in `is facing other` and `is camera on` (both from the AsteroidBee Pictures category).

8. Inside the if's **do** slot: `take pic`.

9. **Predict before simulating.** Ask: "How often CAN this thing shoot?" Walk the math together: the cooldown allows at most one attempt every 3 seconds. But each attempt costs 1.0 energy, and you earn back only +0.5 per second, and **only in the light**. So while the sun shines, you earn 1.5 back during each cooldown and can keep firing; the moment the recharge stops, energy becomes the real limit: five attempts and the tank is empty.

10. **Simulate as Blue** against the default opponent (it runs no code and just sits at its start, 0.8 m away, in the light). Watch the simulation viewer: the satellite turns, and once `is facing other` starts answering yes, the **scoreboard climbs in steps of roughly 2–3 points** every few seconds. Celebrate: first points of the season.

11. **Let the t = 60 trap spring.** Don't warn them. At t = 60 the Light and Dark Zones **swap**, and the idle opponent is suddenly sitting in the **dark**, where a camera can't see it. Photos now fail, but each one still costs 1.0 energy, and our satellite (also in the dark now) isn't recharging. Let students notice the score has stalled while the camera keeps clicking. Ask: "Is the camera broken? Check the rules: what happens to a photo of someone in the Dark Zone, and what did each of those failed shots cost us?"

12. **Refine to version 2.** Extend the if's condition with two more `[a] and ▾ [b]` blocks: add `check in light <me/other> ▾` set to **other** ("only shoot when the target is in the light"), and a compare block `[a] == ▾ [b]` with its dropdown set to **>**, holding `get <my/other> ▾ energy` (set to **my**) and a number block with **1.5** ("keep an energy reserve"). Re-simulate: during the dark stretch the camera goes silent, energy holds, and shooting resumes when the lights swap back at t = 150. No more wasted shots.

13. **Show the C.** Flip the **"C Code" toggle**: the whole guard is one `if` with the four questions chained by `&&`. Point at `void loop()` and remind them the simulator calls it once per second: the `if` is asked 180 times, and that is the entire trick.

The mistakes you will see

1. The timeline trap: "turn toward them, THEN take the picture"

- **What it looks like:** `take pic` stacked directly under the aiming blocks with no `if` around it: two statement blocks in a row, like steps in a recipe. In the simulation, the camera fires from the very first seconds while the satellite is still rotating; almost every shot fails, each failure burns 1.0 energy, and the score barely moves.
- **Why it happens:** turning to face the opponent takes several seconds, but the page does not wait: every block on it runs every second, so `take pic` fires long before the aim is true. The if guard, re-asked every second, is what "waits."
- **The question to ask:** "Which block is your program's way of waiting? (Hint: it is not a movement block.)"

2. No energy guard: "the camera broke"

- **What it looks like:** the camera fires every 3 seconds no matter what; sometime mid-match it simply stops scoring, and the student reports the camera is broken.
- **Why it happens:** every attempt costs 1.0 energy (success or failure), and the recharge is only +0.5 per second, and only in the Light Zone. Once the satellite stops recharging, a few attempts drain the tank to zero, and a satellite at 0 energy can't take pictures at all.
- **The question to ask:** "What does each attempt cost, and how fast do you earn it back, and only where?"

3. Shooting at a hidden target after the t = 60 swap

- **What it looks like:** version 1 works beautifully for the first minute, then the score flatlines from t = 60 onward while the camera keeps clicking and energy drains away.
- **Why it happens:** at t = 60 the Light and Dark halves of the arena swap, so the idle opponent is now in the Dark Zone, invisible to the camera. The program never asks where the target is, so it keeps paying full price for photos of darkness.
- **The question to ask:** "Where is the opponent after the lights switch? Can a camera see into the Dark Zone?"

Reference solution at a glance

Init page: a single array declare block in the "global variables" slot: float **aim**, length **3**.

Main page, top to bottom in the "loop" slot: `get att to other to [aim]` re-measures the direction to the opponent; `set AttitudeTarget` (the dropdown on `set PositionTarget ▾`) points the satellite along **aim**; then one `if … then …` block whose condition chains four questions with `[a] and ▾ [b]` blocks (`is facing other`, `is camera on`, `check in light other`, and `get my energy` **> 1.5**), with a single `take pic` inside the **do** slot. Four-and-a-bit blocks, re-run once per second, and that's the whole spy camera.

The full layout and the exact generated C are in reference-solution.md.

Wrap-up questions

1. There is no "wait" block anywhere in our program, yet it clearly waits: for the turn to finish, for the cooldown to end, for the opponent to step into the light. What is doing the waiting for us?
2. Why is it sometimes smarter NOT to take a picture even when the camera is ready? What does a failed attempt cost, and what does it earn?
3. Our opponent today never moved. Which parts of our program would already work against a moving opponent, and why does re-running the aiming blocks every second make that true?