Assignment 6: Power and Pickups — Teacher Guide

At a glance

Learning objective	Students run a real mission plan on the state machine: collect a score item (+1.5), then an energy item, then park, learning item pickup mechanics (park within 5 cm, slower than 0.01 m/s), whole-array position targets, and the light/dark energy economy.
Time estimate	90 minutes
Project setup	New project on the **AsteroidBee 2026** game, **Graphical Editor**
Prerequisite assignment	Assignments 1–5
Blocks introduced	`get item _ loc to` (AsteroidBee MS- Items), `set PositionTarget` with a whole array, `check have item _ me`, `get item _ type`; `check in light me` / `get light switch time` for discussion

The one idea this assignment teaches

The game itself can answer the "is this step finished?" question. In Assignment 4, students built a multi-step plan where each step ended when a coordinate check said "close enough." Today the plan has the same three-step shape, but the test that advances a step is no longer arithmetic the students wrote; it is the game's own confirmation: `check have item 4 me`. The destination is also no longer numbers the students typed: the block `get item 4 loc to` fills a 3-slot array with the item's location, and that whole array is handed to `set PositionTarget`. The students' program asks the game where to go and whether it has succeeded, both questions asked fresh, every second.

The second idea hiding inside the first: **picking up an item is not an action block.** There is no "grab" button. The rule, straight from the game's source code, is: be within 5 cm of the item and be moving slower than 0.01 m/s. In plain words: park on it. The position controller students have used since Assignment 2 does exactly that when you give it the item's location and wait. So the pickup happens as a consequence of per-second position control, and the program finds out it happened by asking.

> ⏱ **Every second:** the main page re-runs from the top. It checks which step the plan is on, re-fills the `dest` array with the item's location, re-issues `set PositionTarget`, and asks the game `check have item 4 me`. Nothing in the program "waits"; the waiting you see on screen is just the same question getting the answer "no" many seconds in a row, until one second the answer is "yes," the step variable changes, and the next second a different branch runs.

This is the per-cycle pattern from Assignment 4 doing real game work for the first time: remember the step in a variable, act on the step every second, advance the step when its goal is met, except the goal is now confirmed by the game, not computed by the student.

Before class

- [ ] Read (or re-read) how-blocks-run.md; the "everything runs every second" idea carries this whole lesson.
- [ ] Skim the **Items** and **Light and Dark Zones** sections of game-rules.md. Know these by heart: pickup = within 5 cm and slower than 0.01 m/s; score items 4/5/6 (the squares on the arena map) are +1.5 points each; energy packs 0–3 refill energy to 5.0; the light/dark halves swap at t = 60 s, and again at t = 150 s.
- [ ] Build the reference solution yourself once (see reference-solution.md) and simulate it, so you know what the log lines and the +1.5 score jump look like.
- [ ] Have the arena map ready to project: ../images/arena-diagram.svg.

- [ ] Each student/pair needs a new project on the AsteroidBee 2026 game with the Graphical Editor. (Their Assignment 4–5 projects can stay open in another tab for reference; today's program has the same three-step shape.)
- [ ] Print one printable.md packet per student.

Walkthrough: what to say and do

1. **Brief the mission (project the arena map, ../images/arena-diagram.svg).** Say: "Nine pieces of debris are floating at fixed, known spots. The score items (squares on the map), items 4, 5, and 6, are worth **+1.5 points each**, and right now they're sitting in the Dark half of the arena. The energy packs (circles on the map), items 0 through 3, refill your energy to full. The blue ones, 7 and 8, are mirrors; those are next week's assignment. Today's shopping list: score item 4 at (0, 0.6), then energy pack 0 at (0.25, −0.40), then come home and park."

2. **Teach the pickup rule.** Say: "There is no 'grab' block. The game's source code says you collect an item when two things are true at the same time: you are within **5 centimeters** of it, and you are moving slower than **1 centimeter per second**. In plain words: **park on it.**" Ask the class: "What do we already own that parks the satellite at an exact spot?" (They should land on the position controller: `setPos _, _, _` and friends fly to a point and stop there.) Punchline: "So pickup needs no new trick. Give the controller the item's location, and wait."

3. **Demonstrate the new tool: the game fills your array.** On the main page, drag `get item 4 loc to` from **AsteroidBee MS- Items**. Point out it comes with a whole-array block pre-attached; pick the array's name from its dropdown (we'll create `dest` in a moment). Say: "Every second, this block writes item 4's location into the three slots of `dest`. We never type those numbers."

4. **Demonstrate the whole-array destination.** Drag `set PositionTarget` from SPHERES Controls and plug the same `dest` array into it. Say: "Until today we typed coordinates into `setPos _, _, _`. This is the first time we hand the movement system a **whole array**: the game filled it, we just pass it along." Ask: "Why might letting the game fill in the destination be better than copying numbers off the map?" (No typos; and later, destinations that we can choose while the match is running.)

5. **Set up the init page.** Switch to init. In the "global variables" slot: an array declare block (`float`, name `dest`, length **3**) and a declare block (`int`, name `step`, initial value **0**). Mention: "No `myState` array needed today; we are not doing coordinate math. The game answers our questions instead." (Students may add one anyway; harmless.)

6. **Say the plan in words before building it.** Write on the board:

7. **Step 0**: park on score item 4, until `check have item 4 me` says yes.
8. **Step 1**: park on energy item 0, until collected.
9. **Step 2**: hold position at (0.25, −0.2).

Ask: "What does this plan look like as blocks? We built this exact shape in Assignment 4." (Three `if` blocks checking `step`, each with its own work and its own "am I done?" test.)

1. **Build step 0 together** on the main page, inside the "loop" slot: an `if … then …` block whose test is a compare block, `step = 0`. Inside it: `get item 4 loc to [dest]`, then `set PositionTarget [dest]`, then a second `if … then …` whose test is `check have item 4 me` (from AsteroidBee MS- Items; set the dropdowns to item **4** and **me**). Inside that inner `if`: `set step = [1]` and `DEBUG ["Score item collected!"]`. Say it out loud as a per-second script: "Every second: if I'm on step 0, refresh the destination, re-issue it, and ask the game: is item 4 mine yet? The first second the answer is yes, the step changes."

2. **Point at what's different from Assignment 4.** Say: "Same machine, new finish line. In Assignment 4 we decided we'd arrived by checking coordinates. Today the **game** tells us the step is finished. Position is not the goal; possession is."

3. **Students build steps 1 and 2 themselves.** Step 1 is a copy of step 0 with the item number changed to **0** in both the location block and the check block, advancing to step 2 with a `DEBUG ["Energy topped up!"]`. Step 2 is a single `setPos 0.25, -0.2, 0` inside `if step = 2`. Circulate. The classic mistakes below will appear now.

4. **Tell the energy story while they simulate.** Say: "Item 4 is in the Dark half. While you're there your solar panels get nothing (no recharge), but the opponent's camera can't photograph you either. Dark is safe but expensive. And the halves **swap at t = 60, and again at t = 150**." Show that the palette has `check in light me` and `get light switch time` for programs that want to know; we'll use them when strategy matters. (Energy facts: recharge is +0.5 per second in the Light Zone only; thrusters cost almost nothing; pictures are what really spend energy.)

5. **Simulate the full 180 s as Blue against an idle opponent.** Watch together: the satellite crosses into the dark half, settles onto item 4 (**score jumps +1.5** and the first DEBUG line prints), then travels to item 0, energy snaps back to a full 5.0, second DEBUG line, then it parks at (0.25, −0.2) for the rest of the match.

6. **Discussion to close:** "In a real match, what happens to this plan if the OPPONENT grabs item 4 first?" Let them reason it out: `check have item 4 me` will answer "no" every second forever; the plan waits at step 0 for the rest of the match. The fix is an extension in the reference solution: also advance when `not check have item 4 no-one`, meaning "somebody has it, and it isn't going to be me. Move on."

The mistakes you will see

1. Drive-by pickup

- **What it looks like:** The satellite flies right over the item (sometimes straight through it) and nothing happens. No log line, no points. Often the student aimed past the item, or their plan moved on to the next destination too quickly and expected to scoop the item up mid-flight.
- **Why it happens:** Pickup requires parked-slow closeness: within 5 cm **and** moving slower than 0.01 m/s. A satellite in transit fails the speed test no matter how perfectly it crosses the spot. Underneath it is also the leftover timeline instinct ("my blocks said go there, so I went there, so I got it"): blocks as a story that completes, instead of a destination that's re-issued every second until the satellite actually settles.
- **The question to ask:** "Did you stop ON it, or just visit? What two conditions does the pickup rule name?"

2. No advance condition

- **What it looks like:** The satellite flies to item 4, parks, collects it (score +1.5!) and then sits on the empty spot for the rest of the match. The plan never reaches step 1. Or a near-miss version: the student advances on an Assignment-3-style coordinate check, which says "I'm there!" a moment before the pickup actually registers, so the plan leaves without the item.
- **Why it happens:** The movement part of the step got built, but nothing asks `check have item 4 me`, so `step` never changes, and every second the program re-reads the same step-0 branch. Or the student reused the coordinate test from earlier assignments out of habit, and position is not the same thing as possession.
- **The question to ask:** "What EXACT question tells you step 0 is finished: your position, or the game saying the item is yours?"

3. Wiring the wrong array

- **What it looks like:** The satellite sits perfectly still all match (the student fed `myState` into `set PositionTarget`; the satellite is chasing its own position, a destination it has already reached every second). Or: the item location goes into `dest`, but a different array gets plugged into `set PositionTarget`, so the satellite flies to (0, 0, 0) or wherever that other array points.
- **Why it happens:** Whole-array blocks all look alike, and the array name lives in a small dropdown. This is the first assignment where data flows through an array, from a game block into a movement block, so the "same array on both ends" requirement is brand new.
- **The question to ask:** "Which array did the item location go INTO, and which array did you hand to the movement block?"

Reference solution at a glance

On the **init page**, the "global variables" slot holds two declarations: a float array `dest` of length 3 (the destination the game will fill in), and an int `step` starting at 0 (the plan's memory).

On the **main page**, the "loop" slot holds three `if` blocks in a row: the same state-machine shape as Assignment 4, now with a third branch. The `if step = 0` branch fills `dest` with `get item 4 loc to [dest]`, hands it to `set PositionTarget [dest]`, and inside an inner `if` on `check have item 4 me` sets `step` to 1 and prints a DEBUG line. The `if step = 1` branch is identical with item **0** and advances to step 2. The `if step = 2` branch is one block: `setPos 0.25, -0.2, 0`. Every second, the branch matching the current step does its work; the inner `if` is the game-confirmed finish line that moves the plan along. (Fine print: on the one second a pickup registers, `step` changes mid-loop, so the next branch also runs that same second; harmless, it just issues the new destination one second sooner.)

The full block layout, the generated C, and how to verify it are in reference-solution.md.

Wrap-up questions

1. In Assignment 4 we decided a step was finished by checking our coordinates. Today we asked `check have item 4 me` instead. Why is the game's answer a better finish line for a pickup than your own position?
2. Item 4 sits in the Dark half at the start of the match. What does your satellite give up while it's in the dark, and what does it get for free? When do the halves swap?
3. Suppose the opponent grabs item 4 at t = 20 s. What does your program spend the remaining 160 seconds doing, and why? What question could your program ask, every second, to notice and move on?