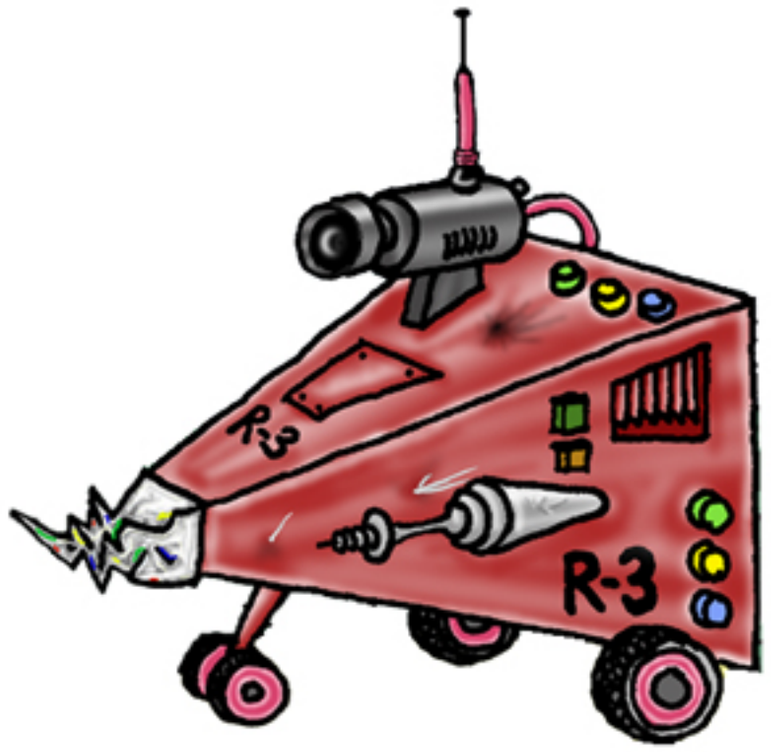
**Kory Heath**
*Game Designer / Unity Developer*

---

## RAMbots




Deep in the heart of The Complex—a sprawling, seemingly infinite maze of gleaming corridors and whirring bots—the Master Priority Scheduler has awakened you for yet another task. You find yourself in a wide metallic chamber, featureless save for the glowing beacons floating near the center. Looks like data-collection again—typically the simplest of tasks. However, as your sensors sweep across the expanse of the chamber, you register the existence of other bots hovering out on the perimeter.

Uh oh. A Multi-Task. Time to check the precedence stack and get going.

But you pause to wonder, as you have so many times before: what *is* this Master Priority Scheduler? What is the Ultimate Task, of which your own is but a tiny, insignificant fraction? How big is this Complex? What's Outside? *Is* there an Outside?

Sadly, the answers to these questions a simple RAMbot is not given to know. All you know is the task that has been set before you: to collect your precious data; to collect it in the proper order; to collect it as quickly as possible.

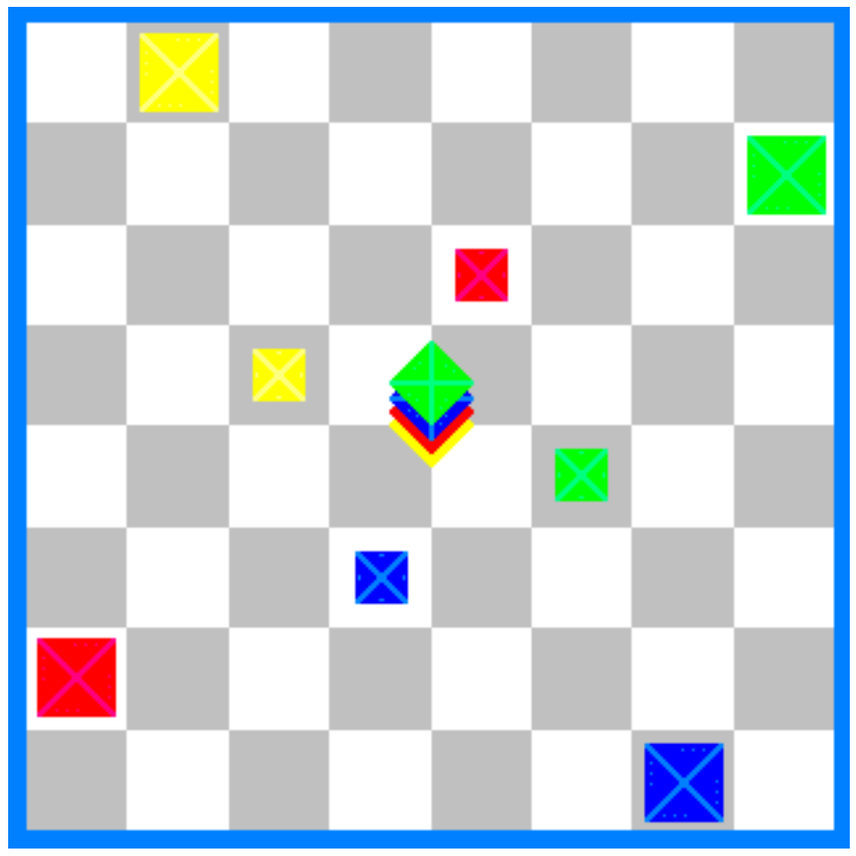
And, of course, to RAM the hell out of anyone who gets in your way.

### What You Need

- Two to four players.
- A Rainbow Treehouse set for each player, plus one extra set.
- A chessboard.
- A cardboard screen for each player.

### Setup

The easiest way to begin setting up RAMbots is to use all of the pieces to build trees of each color. (A “tree” is a small piece stacked on a medium stacked on a large.) Each player should gather an initial “code-pool” of pieces by selecting a single tree of each color. A tree of each color should also be placed on the board. The large pieces on the board represent RAMbots, the medium pieces represent the “precedence stack”, and the small pieces represent beacons. Assign a RAMbot to each player, and place each one upright near its owner, in the spaces shown in the diagram. If there are fewer than four players, set aside any unused RAMbots and code-pool pieces for the duration of the game.



Arrange the four beacons as shown in the diagram, lying down on their sides (it doesn’t matter in which direction), and in random color order. Arrange the four medium pieces into a stack which, from top to bottom, matches the color ordering of the RAMbots on the board, beginning with a random color and moving clockwise around the board. (This ordering is not strictly necessary, but it will make the process of executing programs during the game a bit easier.) The precedence stack should not actually occupy spaces on the board, but it can be placed in the center intersection so that it will be easily visible to all players.

Select one piece of each color from your code-pool (of whatever sizes you choose) and create a “goal stack” *for the player on your left*. This stack will represent the order in which that player needs to tag the four goals to win the game; the top piece the color that needs to be tagged first, and so on. Keep the goal stack you create hidden until all players have created one; then place the goal stack on the corner nearest your neighbor’s RAMbot, on the very edge of the board. Like the precedence stack, these goal stacks do not occupy spaces on the board, and they will not interfere with the movements of RAMbots during the game.

Once the goal stacks have been created, the setup is complete; you’re ready to begin playing.

## Play

A single round of play is divided into two phases: the programming phase, and the execution phase.

### The Programming Phase

At the beginning of this phase, all players set up their screens and begin “programming” their RAMbots. You may lay out up to five of your code-pool pieces behind your screen; they will be executed in order from left to right. Each instruction piece represents a single RAMbot action, which will cause your RAMbot to move and then shoot a beam of colored energy. (See the section entitled “Executing Individual Instructions” for a full description of RAMbot actions.) You are not allowed to set up more than five instructions, but you are allowed to set up less than five.

When all players have finished programming, everyone lifts their screens, and the game moves on to the Execution Phase.

## The Execution Phase

To begin the Execution Phase, each player should slide his or her leftmost instruction forward to the center of the nearest board edge, an action known as “loading the program register”. One of these instructions is about to be executed; to determine which one, first look at the *sizes* of the pieces. Smaller pieces run more quickly than larger pieces, so smaller pieces always take precedence over larger ones. If there is a single smallest piece, simply execute that instruction. (See the section entitled “Executing Individual Instructions” for details on how the individual instructions work.)

If there is a *tie* for smallest piece, compare the *colors* of the tying pieces to the precedence stack in the middle of the board. Higher colors always take precedence over lower ones. If there is a single highest color among the tying pieces, simply execute that instruction.

If there is *another* tie, then the instruction pieces in question must be identical. In this case, compare the colors of the appropriate player’s *RAMbots* to the color precedence stack. The instruction belonging to the player whose RAMbot’s color is highest on the precedence stack should execute first. (This is easy to see, assuming you’ve arranged the precedence stack according to the initial RAMbot colors. The player whose color is currently on top of the precedence stack will have the highest precedence during a tie, and so on clockwise around the table.)

After an instruction has been executed, the instruction piece should be returned to its owner’s code pool, and that player’s next leftmost instruction should *immediately* be pushed forward into his or her program register. Once again, follow the above rules to determine which instruction should be executed next, and then execute it. (Note that it is perfectly possible for a player to execute two or more instructions in a row.) Repeat this process until every instruction has been executed.

When all instructions have been executed, the round of play is over. Remove the top piece from the precedence stack, and place it on the *bottom* of the stack. Now you’re ready to begin another programming phase.

## Executing Individual Instructions

Each instruction in your program will cause your RAMbot to move and then fire a beam of colored energy in front of itself. Each piece in a program should either be lying down and pointing in one of the four cardinal directions, or standing upright on its base. The orientation of the instruction piece indicates how your RAMbot will move, and the color of the instruction piece indicates what kind of beam your RAMbot will fire.

### Movement

An instruction piece lying on its side tells your RAMbot to move in the direction the instruction piece is pointing. When you execute this instruction, move your RAMbot one, two, or three spaces in the appropriate direction, depending on whether the instruction piece is small, medium, or large. If your RAMbot is not already facing in the direction that your instruction piece is pointing, the first unit of the movement instruction will be used to reorient your RAMbot in the appropriate direction. So, for instance, if you have a large movement instruction piece pointing in a different direction than your RAMbot, you must first reorient your RAMbot

(using one unit of the movement action), and then move it two spaces in the appropriate direction. A small-sized movement instruction will therefore reorient your RAMbot without moving it if it's not already facing in the appropriate direction. If at any time your RAMbot is standing on its base (as it is at the beginning of the game), the first unit of a movement instruction will tip your RAMbot down and point it in the appropriate direction, to be followed by the rest of the movement action.

An instruction piece standing on its base represents "reverse gear"—it causes your RAMbot to move *backwards* away from the direction that it's currently pointing, for one, two, or three spaces. If your RAMbot is currently standing on its base, an upright instruction will not move your RAMbot at all.

## Pushing and RAMing

If, during one of your movement instructions, your RAMbot moves forward into or backs into a space that contains a beacon or another RAMbot, the object will be pushed. If there are objects directly on the other side of the pushed object, they will be pushed along with it. If an object cannot be pushed any further (because it's against a wall, or it's against objects which are against a wall), your RAMbot simply stays where it is (though you do still make contact with the object). If your RAMbot runs directly into a wall, nothing special happens.

If you make contact with another object with the *nose* of your own RAMbot, you have RAMed that object. (If you back up into an object, or if you get pushed into another object during someone else's instruction, you do not RAM that object.) If you make contact with another object multiple times during a single instruction, this only counts as a single RAM. Whenever you RAM another player's RAMbot, you cause damage to that RAMbot. Take the highest precedence piece from that player's code-pool and add it to your own code-pool. In other words, take the *smallest* piece available in that player's code pool; if there's a tie for smallest piece, take a piece of the color that's highest on the precedence stack. If there are currently no pieces in that player's code pool, you don't get to steal any pieces. You may not steal pieces from an opponent's currently running program.

If you RAM an *upright* object (whether it's a beacon or a RAMbot), tip it down onto its side, facing away from the point of impact. If that object matches the color currently on top of your goal stack, you have tagged a goal. Remove the top piece from your goal stack, and add it to your own code-pool.

## Colored Energy Beams

After your RAMbot moves, it will fire a colored energy beam in a straight line out in front of itself. The beam's color is determined by the color of your instruction piece. The beam will affect the first object it hits. If a beam hits a wall, or fires straight upwards, it has no effect.

## Blue—Push

A blue beam will push any object it hits (along with anything else that the object runs into) away from your RAMbot for one, two, or three spaces, depending on the size of your blue instruction piece.

## Yellow—Pull

A yellow beam will pull any object it hits toward your RAMbot for one, two, or three spaces, depending on the

size of your instruction piece. If an object is pulled all the way into the nose of your RAMbot, the object remains in the space next to your RAMbot. This does not count as a RAM; you cannot damage another player, tag a goal, or knock over an upright object in this fashion.

## Red—Slide

A red beam will slide any object it hits sideways (along with anything else that the object runs into), away from the beam and towards the center line of the game board, for one, two, or three spaces, depending on the size of your red instruction piece.

## Green—Activate

A green beam will set any object it hits upright.

## Winning

The first player to tag all four goal colors in the order specified by his or her goal stack is the winner.

## Special Notes

*Compilation Caps*—Although it isn't strictly necessary, it's helpful during the programming round to have some kind of method of indicating who's finished programming (so it's immediately obvious when everyone's ready). This can be something as simple as placing a coin on the edge of the board next to your goal stack when you're done. However, we prefer to give each player a small black Icehouse piece. Drop your "compilation cap" on top of your goal stack to indicate that you're compiled and ready to run. (Of course, you're free to "uncompile" and change your program, as long as the execution phase hasn't started yet.)

*Code Pool Disclosure*—Although players may choose to move their code-pools behind their screens during the programming phase (to keep people from seeing which pieces they're using), the official rule is that you're allowed to know exactly what pieces each player has to work with. Therefore, it's legal to ask players about their pieces (how many small pieces they have, what colors they are, etc.) at any time during the game, and they must answer honestly.

*Speedbots*—For a fast-paced variation of RAMbots, try this rule: the programming round ends when all but one of the players have compiled; the uncompiled player must use his or her program as it lies.

## Links

- Design History
- RAMbots at BoardGameGeek

## Acknowledgments


- Game Design—Kory Heath
- Game Development—John Cooper, Jacob Davenport, Kristin Matherly

- Playtesting—Andrew Plotkin, Andrew Looney, Alison Frane, Dale Newfield, Dan Efran, Peter Hammond, Margit Gedra

---