Overview: This document provides analysis and suggested solutions to the problems posed in the AgentCubes Online Troubleshooting Session Guide document and associated example AgentCubes Online projects. Each section includes the original problem description, the link to the associated AgentCubes Online project, suggested solution(s), and, where appropriate, the link to the AgentCubes Online project that implements the suggested solution.

1) Frog will not move in certain directions: Run the simulation at the link below and move the frog using the arrow keys. What happens? Examine the frog agent behavior. What is the problem? How would you correct it?

https://www.agentcubesonline.com/project/59472

ANALYSIS AND SUGGESTED SOLUTION: In this example, the frog will only move up and down in response to keyboard input. As can be seen in the screen shot below, there is a blank rule among the rules that respond to keyboard input. AgentCubes examines the rules of each agent from first to last. The first rule with all of its conditions satisfied will be executed; all rules below that rule will not be examined. Since a blank rule has no conditions, it is unconditional, which means that it will be selected for execution, even though there are not actions in the rule. Thus, all rules below the blank rule in the example problem, which includes two rules responding to the left and right arrow keys, will be ignored. Therefore, the frog will not move in those two directions.
2) **Cars get stuck in front of the right tunnel:** Run the simulation at the link below. In this example, the cars get stuck and “queue” up in front of the right tunnel. *Why is this occurring? How would you correct the problem?*

[https://www.agentcubesonline.com/project/59570](https://www.agentcubesonline.com/project/59570)

**ANALYSIS AND SUGGESTED SOLUTION:** In this example, the cars “back up” against the right tunnel, rather than disappear (be absorbed by) the tunnel. Absorption requires a rule in which one agent is touching another (that is, are next to each other or one on top of the other), at which point the conditions of the rule are satisfied and one agent is erased. The erase action can be located in the behavior of either agent; it is a designer’s choice. In this case, since the car is doing the moving, it seems reasonable for the car to take the action, which is to remove (erase) itself from the world. As can be seen in the screen shot below, there is no rule in the car behavior of the original project to perform the absorption:
Below is an example of an absorption rule in the car behavior:

Each of the rules is conditioned upon the car “seeing” either the road or the tunnel to its right; if the road is to the right, the car moves to the right; if the tunnel is to the right, the car erases itself.
3) **Cars get stuck on right tunnel:** Run the simulation at the link below. In this example, the cars pile up on top of each other on the right tunnel. *Why is this happening? How would you correct this?*

https://www.agentcubesonline.com/project/59571

**ANALYSIS AND SUGGESTED SOLUTION:** This example is similar to the previous example, in that there is no absorption rule for the car to erase itself when it “sees” the tunnel to its right. In addition, there is also a missing condition in the movement rule that requires the car to “see” the road in order to move to the right, as shown below. As a consequence, the car moves unconditionally to the right until it reaches the end of the world space, which results in cars stacking on top of each other.
The solution is to implement the absorption rule described in the previous example, and for completeness, add the additional condition described above, as shown below:

![Behavior: car](image)

Note that even with these rules implemented as shown above, it is still possible for cars to back up before the right tunnel. This can happen if the user has saved the world with at least one car already stacked on the right tunnel. In that case, the leading car will not “see” the tunnel; rather it will see the car stacked on the right tunnel. The “see” condition (as well as the “see a” and “next” conditions) refers to the topmost agent if agents are stacked. Even though the viewer, in birds-eye view” can see a tunnel beneath the car, from a programming view, only the topmost agent in a stack is visible to surrounding agents.
4) **Repeated sound and message displayed**: Run the simulation at the link below. Move the frog in front of a car. After the honk sound and the dialog is displayed, tap “OK” in the dialog window. *What happens? Why is this happening? How would you correct the problem?* NOTE: In order to clear the problem, you will use your keyboard and mouse* at the same time: press the “Enter” key while you simultaneously tap the “stop” button on the tool bar; you may have to do this multiple times!  
* You may have a trackpad or other device instead of a mouse.

https://www.agentcubesonline.com/project/59572

**ANALYSIS AND SUGGESTED SOLUTION**: In this example, when the sound plays and the dialog message appears, the user taps the “OK” button and the sequence repeats. Examine the behavior below:
As can be seen, the behavior is to play the sound and show the message when the frog sees a car to its left. After the user taps the “OK” button, this condition is still true. As a consequence the actions repeat. Programmers refer to this as a loop. Unfortunately, it may be difficult to terminate this loop, as it requires responding to the dialog (by tapping “OK” or by pressing the “Enter” key) and almost simultaneously tapping the “stop” button to stop the simulation. The programming solution is to place a simulation ending action – either stopping the simulation or reloading the world – after the “show message” action, as shown below:
5) **Continuous sound**: Run the simulation at the link below. You will hear a continuous honking sound. The world viewport may go completely black. *Why is this happening? How do you think this occurred? How would you correct it?* NOTE: You may need to reload the web page in order to stop the sound.

https://www.agentcubesonline.com/project/59573

**ANALYSIS AND SUGGESTED SOLUTION**: In this example, the problem is not a programming issue. Rather, it is a problem with a “stray” frog in the world, in this case buried under a road agent as shown below:

![Frog agent under the road.](image)

It is likely that the user would detect this construction, at least in a simple world like the example. However, in a large and complex world, as well as one with agents which are not as three-dimensional in nature, detection may be more difficult. In such cases, it may be necessary to “peel” layers of agents a little at a time to find stray agents causing the problem.
6) **Too many turtles – Reason A:** Run the simulation at the link below. Notice the number of turtles emerging from the island. *Why are there so many? How can you correct this problem most efficiently?*

https://www.agentcubesonline.com/project/59575

**ANALYSIS AND SUGGESTED SOLUTION:** In this example, buried agents are the cause of the problem, which is turtles that almost continuously, rather than randomly, emerge from the island on the river. Since the island agents are relatively flat in terms of 3-D profile, it is difficult to see that they are stacked, as shown in the screen shot below:

Since each agent operates independently, and since they all “see” the river agent to their left, as a turtle moves to the left, the river agent is exposed, so that at least one of the islands “sees” a river agent and, as a consequence, may generate a new turtle. With several island agents, a new turtle is likely to be generated as soon as the previously generated turtle moves.
One question that naturally arises is how the generator agents get stacked in the world. One typical way is that the user incorrectly programs the generator to make a new generator, rather than a turtle, and also neglects to change the direction of generation; the default of the “new” action is the “dot” or “in place” location. Thus, it is possible for the generator agent to make a new one of itself. If the user allows this to continue and then saves the world before reloading it, the result is stacked generator agents.

The solution, of course, is to remove all but one of the generators. Sometimes, if there are dozens of agents—an unknown number—this can be tedious. A more elegant approach is to have AgentCubes do the work by temporarily inserting a rule in the generator agent’s behavior to erase itself if it is stacked above a generator agent. Then, single stepping the simulation will result in all but one of the generator agents being erased. Afterward, the user would remove the temporary rule and SAVE the world! Below is an implementation of this approach:
7) **Too many turtles – Reason B:** Run the simulation at the link below. Notice the number of turtles. *Why are there so many? How can you correct this problem most efficiently?*

https://www.agentcubesonline.com/project/59654

**ANALYSIS AND SUGGESTED SOLUTION:** In this example, too many turtles are being generated in the river, but it’s not because of multiple island agents stacked above each other. Rather, there are island agents buried under the river, as can be seen below:
Buried agents can sometimes be difficult, particularly when they are tiles or flattened inflatable icons. However, since it is a common problem, especially for beginning designers, it is worth an initial suspicion. In this case, there are two island (generator) agents that are visible when the river agents above them are removed. As in the previous example, it is possible to have AgentCubes remove the agents with some temporary programming and a single-step play, as shown below:

Note that you may need to specify “somewhere below” instead of “immediately below” if you suspect that there are stacked generator agents under one or more river agents. If you use this technique, use the single-step function to accomplish the agent removal, then be sure to save the world afterward.
8) **Too many turtles – Reason C:** Run the simulation at the link below. *Why are there turtles in the upper log lane? How can you correct this problem most efficiently?*

https://www.agentcubesonline.com/project/59652

**ANALYSIS AND SUGGESTED SOLUTION:** In this example, as in the previous examples, there is a buried generator agent. However, this one is buried under a river agent, which is under a tree agent, which is the log generator. As a consequence, turtles and logs appear in the same channel, as shown in the upper river channel. See below:
As with previous examples, it is possible to use AgentCubes itself to correct the situation by removing the buried turtle generator (island) agent, as shown below:

As discussed in previous examples, be sure to save the world after the buried agents are removed.
9) **Too many turtles – Reason D:** Run the simulation at the link below. Notice the number of turtles emerging from the island. *Why are there so many? How can you correct this problem?*

https://www.agentcubesonline.com/project/59655

**ANALYSIS AND SUGGESTED SOLUTION:** In this example, the problem is not with the world construction, but with the behavior of the turtle generator (island agent). As indicated below, the order of the conditions is the source of the problem. To understand this it is first important to understand how conditions are processed. As with rules, conditions within a rule are evaluated in order from the first condition listed. In many cases, the evaluation order does not significantly affect the result. However, in the case of the combination of timing (“once every” condition) and probability (“% chance” condition), condition order is critically important. AgentCubes evaluates the rules of every agent continuously. Given today’s computer processing speeds, this means that each agent’s rules may be evaluated many times per second. The “% chance” condition determines the probability that on any given cycle through an agent’s rules that condition will be true. If the value of the condition is 50%, this means that about every other time that agent’s rule is evaluated the condition will be true. Now consider the timing event. If that event is listed first that means as each time interval passes, about 50% of the time the condition will be true. So, if the “once every” value is 1 second, this means that about every 2 seconds the combined conditions will be true on average. That is what is desired.

Now suppose the conditions are reversed. The “% chance” condition will be evaluated many times per second. The timer condition will be true once per specified time interval, say 1 second. Suppose that the rule is evaluated 20 times per second. This means that on the average about every 0.1 second the “% chance” condition will be true. When the “once every” timer event becomes true, this means that within about 0.1 second, both conditions of the rule will be true. Adding the two values together, this means that the actions of the rule will be performed about every 1.1 seconds, which is much more frequently than desired.
The solution to this problem is simply to reverse the order of the conditions:

This order of conditions results in essentially a new turtle every second.

Proper order of conditions.
10) **Clock problems:** Before running the simulation at the link below, examine the behavior of each agent (Light, Light3, and Mover). Decide the behavior you expect; then run the simulation. *Is the behavior as expected? Why not? How would you change the behavior to accomplish what appears to be intended?*

https://www.agentcubesonline.com/project/59657

**ANALYSIS AND SUGGESTED SOLUTION:** These are examples of the problems that can arise when using multiple timer (“once every”) conditions in an agent’s behavior. From the behavior in each agent, it is apparent that the designer wanted the colors to “toggle” back and forth at the “once every” interval. The intent of the mover agent was to move back and forth, like the pendulum of a clock. However, that does not happen. Instead, the colors seem to “freeze” and the mover agent moves to the left and stays there.

It is important to understand that timer events are managed for each agent separately. When the condition’s specified time interval expires, that condition becomes true for only that evaluation cycle for that agent’s rules. If there are multiple rules with the “once every” condition as the first condition in those rules, the result could be that only the first rule “sees” the timer event expire. Look at the behaviors of the agents in this example:

```
while-running

if once-every 1 sec then change ●

if once-every 1 sec then change ○
```

*When timer event in first rule 'expires', Light changes to green and resets the timer. So the second rule is never evaluated.*
In each of these agents’ rules the first condition in each rule is the timer (“once every”) condition. Since the timer interval values are the same, the second rule in each group will never be executed. There are multiple ways to address this problem. In the case of the Light agents, one can use the state of the color to govern which rule will be performed. After adding conditions to check the color state, the timer event will be evaluated only if the agent has the proper color. Here is the corrected behavior for the Light agent:

In this solution, using the ‘see’ condition causes the appropriate rule to be selected before the timer event is evaluated. The agent color acts as information -- like an agent attribute -- that controls the condition evaluation. When the agent changes color, the next time rules are evaluated the other rule is evaluated, so that the timer event will apply to that rule.
Another approach is to use only one timer event in the “While Running” method and delegate the color change decision to a separate, subordinate, method. Note that only rules in the “While Running” method of an agent are continuously evaluated. Rules in all other methods are evaluated only when that method is called upon. In the example below, there is only one rule in the Light3 agent’s “While Running” method; that rule has the timer condition. When that timer event occurs, the rule calls upon the rules in the “Decide” method to determine which color to change. Thus, each time interval, the color will alternate, which is the desired behavior.

Note that, in effect, the color (each color is a different shape of the Light3 agent) is an attribute of the agent; that is, it is a piece of information that helps distinguish between the two different shapes. Agent shape is used for decision making. This will not work in the case of the Mover and Mover2 agents, as will be discussed next.

The Mover and Mover2 agents do not have multiple shapes. So there is no inherent attribute – shape – to use for decision making. The solution in this case is to create an attribute – an agent attribute, in this case “direction” – to use for decision making. By assigning a value of 0 or 1 to this attribute, one can base a decision on the attribute value.

As with the Light and Light3 examples, it is possible to implement a solution with all the rules in the “While Running” method, or by using a separate subordinate method to separate the timer condition from the rest of the actions. Below are examples of these two approaches.
The Mover agents have only one shape, so different colors cannot be used for decision making, as with the Light agents. One way to solve this is to use an agent attribute that ‘toggles’ between two values in order to determine which direction to move. As with the Light agent solution, this approach uses the agent attribute value to determine which rule in the While Running method will be evaluated, so that the timer event will apply to the appropriate rule. At each event, the agent attribute is toggled so that the other rule will be executed on the next timer event.

As with the alternative solution for the Light agent, the While Running method can be used to manage the timer and a separate method can be used to manage the decision based on the agent attribute.
11) **Super frog is not super**: Before running the simulation at the link below, examine the behavior of the Super_frog agent. Verify that the Super_frog’s rules should allow it to move anywhere in the world without a problem. Now test the Super_frog, as follows: erase the Frog from the world, add a Super_frog in its place with the pencil tool, then run the simulation and move the Super_frog throughout the world using the arrow keys.

Next, stop the simulation and reload the world. Run the simulation and use the arrow keys to move the Frog agent to the far right of the bottom row, where being on top of the Power_up agent should cause it to change to a Super_frog. Finally, navigate the Super_frog agent to the goal. *What happens? Why does it not behave as expected? What would you change to make this work?*

https://www.agentcubesonline.com/project/59660
ANALYSIS AND SUGGESTED SOLUTION: Examining the behavior of the Frog agent, one can see the intent of the designer: when the Frog moves onto the Power_Up agent, the Frog should become a Super_frog. However, a little experimentation shows that doesn’t happen. The answer lies in the behavior of the Frog:

Changing to the shape of another agent does not change into that agent. It is like wearing a mask of someone else: you are not that other person! As a consequence, the agent is in a 'limbo' state - neither the frog nor the super-frog, so it cannot respond to the keyboard commands.

As noted in the image, the Change action does not change one agent into another; rather, it is used to change among shapes within a single agent. The result of the rule in this illustration is that the Frog agent may take on the appearance of the Super_frog, but will not have its behavior. Consequently, the Frog will not behave like a Super_frog. The solution is to erase the Frog agent and replace it with the Super_frog agent, as shown in the image below:

If you want an agent to be 'transformed' into a different agent, you must create the other agent ('new' action) and delete the original agent ('erase' action).
12. Super_frog win error: Run the simulation at the link below. Move the Frog onto the Power_up agent to transform it into a Super_frog agent. Then move the Super_frog agent to the goal to win. What happens? Why? What change is required to make this work?

https://www.agentcubesonline.com/project/59662

ANALYSIS AND SUGGESTED SOLUTION: In the Super_frog behavior, there is a method to handle the winning situation. A cursory review of the behavior may not reveal the source of the problem: misspelling!

As indicated in the image, both win_0 and win_o exist as names for method. Most likely, the designer created one name in either the message action or in creating a new method. Then, rather than select that name in the corresponding method or message, the designer created a second name, but slightly misspelled it. Spelling counts! The solution is to choose one of these names for both the message and the method.
Message and method names match!
13) **Simulation does not end:** Before running the simulation at the link below, select the Timer agent in the world and open the Agent Attributes window. Run the simulation at the link below. The attribute should be incremented to 10, at which point the simulation should end. *What happens and why?*

https://www.agentcubesonline.com/project/59665
ANALYSIS AND SUGGESTED SOLUTION: This is another case where spelling gets the designer into difficulty. Here, the designer used a lower case letter “l” in one place and the number “1” in another:

In this case, there are two different spellings of the agent attribute. An attribute's value is initialized to zero when first referenced. In this case, clock_1 is zero and Clock_L is set to clock_1 + 1 repeatedly; the intent was to increment Clock_L repeatedly.

Unlike with methods and messages, agent attributes can become “uncoupled” when defining rules. This is because the operand fields of the “test” condition and the “to” operand field of the “set” action are free-form; that is, they do not use selections from menus. Therefore, there is a greater possibility for spelling variable names differently. Especially with certain fonts, in which lower case “l” looks much like the number “1” (as in this document), it can be very easy to inadvertently misspell an agent attribute, for example. One technique to minimize this kind of error is to use variable names that have unambiguous characters. For example, where possible do not use the numbers “0” or “1” in names, since they can easily be mis-labeled as the upper case letter “O” and the lower case letter “l”. In this specific case, since variable names are already defined, it is straightforward to change all of the references to be the same:
The solution is to spell the agent attribute the same way in all references!
14) **Backward operation – Part 1 of 2**: Run the simulation at the link below using the “standard world” world, which should look like the first image below. Move the frog up to the river to verify that the simulation works as expected. Next, select “new world_2”, which should look like the second image below. Run the simulation and note that the arrow keys seem to work backward. Why? What do you think the designer did to cause this? How can this be corrected?

https://www.agentcubesonline.com/project/60328
ANALYSIS AND SUGGESTED SOLUTION: This is a classic error with AgentCubes Online (as well as the computer-resident version of AgentCubes), particularly for beginners. In this example, the arrow keys seem to work backwards: up is down, left is right, etc. What the designer did was to experiment with the camera control tools, in particular the rotation tool, to rotate the world 180 degrees. Then the designer placed agents in the world. With the exception of the Frog agent, since the other agents in the world are symmetrical, it is not possible to see this. Placing a second Frog agent on the world will demonstrate, however, that, in fact, the world is rotated. No doubt, the designer saw this and then used a “rotate” action to rotate the Frog 180 degrees before saving the world.

Unfortunately, there is no easy “fix” for this, short of reconstructing the world, because the agents are all in the opposite order from what the designer planned. Rotating the world as it is 180 degrees will demonstrate that the Frog agent moves properly; however, the highway is above the river, rather than below. This is the kind of error that a student usually makes only once! Note that simply clearing the world does NOT change the rotational orientation. Rather, it simply removes all agents. Therefore, the designer must clear the world, then correct the rotation before placing agents. Another approach is to simply create a new world and delete the original one.
15) **Backward operation – Part 2 of 2**: Using the same project as in the previous example, select “new world_1” and attempt to place any agent into the world. What happens? Why? What do you think the designer did to cause the problem? How can it be corrected?

**ANALYSIS AND SUGGESTED SOLUTION**: This example is similar to the previous one. However, it is different in that the user not only rotated the world, but also flipped it upside down. The consequence of this is that one cannot actually place agents in the world. As with the previous example, the solution is either to clear the world and re-rotate it before placing agents, or to create a new world and delete the original one.