# G54GAM – Lab Session 5

## Extending our tank game

This lab session is based on extending the two-player, split-screen tank game that we began making last week. If you didn't complete the exercises last week then feel free to carry on from where you left off.

Your tank game should now have the following components:

- A large room with walls around the edges and some walls that provide features
- A split-screen view that shows two different regions of the arena at the same time
- Two steerable tanks that can be driven around the arena by two players using two different sets of keys

The aim of this week's session is to add a more advanced kind of game play using a **transitive relationship**, where each player has to try and find better weapons for their tank using objects that can be collected, and then to create a basic artificial intelligence system to control one of the tanks using a simple **finite state machine**.

## Collectable resources

We're going to make a number of objects available to the players that can be collected to give the tanks better weapons, which make it easier to destroy the opponent. However, to retain **balance**, each new weapon must have an associated cost – to achieve this we're going to give the better weapons limited ammunition, and have them appear sporadically in the room, so the players have to search and compete with each other for them. When a player picks up a new weapon, they lose their old one.

We'll have four different resources (these are just examples; you can implement different resources if you like)

- Basic gun – each player starts with this, and it has unlimited ammunition. It fires a bullet in a straight line and blows up when it hits a wall
- Medium gun – has 50 bullets, and does more damage to the opponent, appears quite often
- Big gun – has 20 bullets and does a lot of damage to the opponent, appears rarely

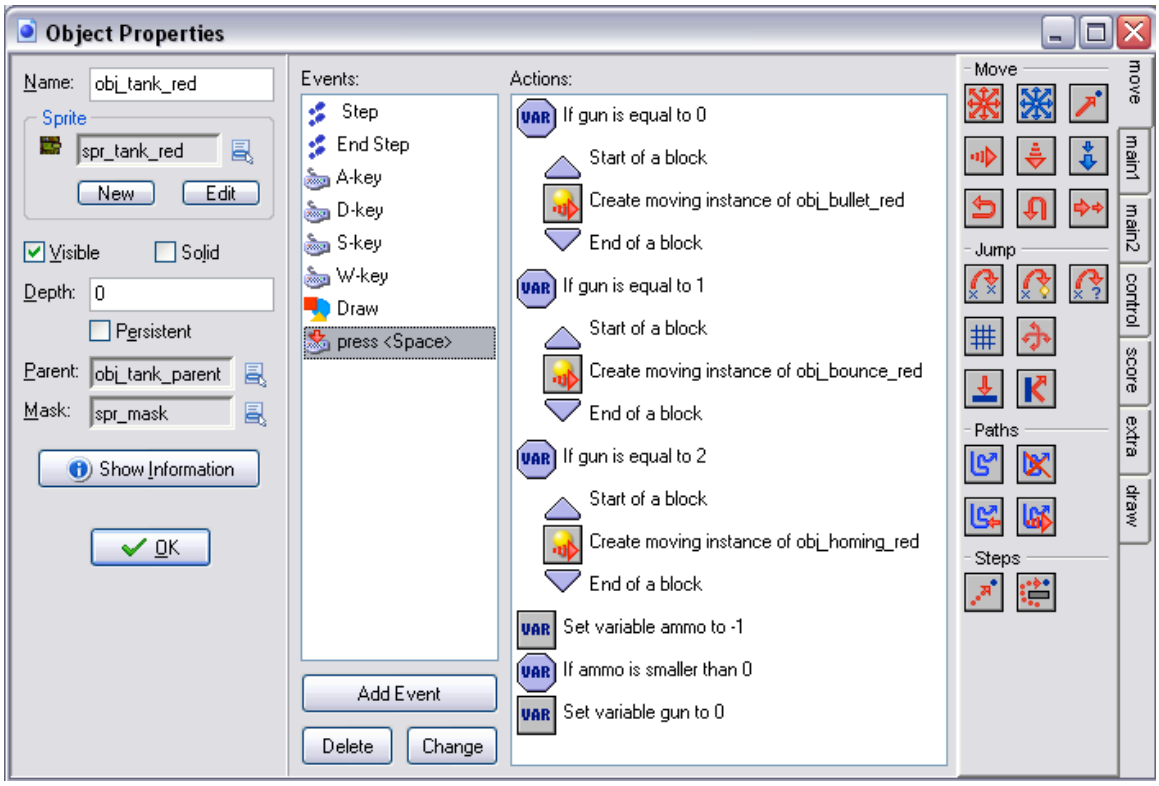
- Health – appears quite often, and restores the tank's health to its full value

This is actually quite straight-forward to implement. The tank object needs to have two instance variables to handle which gun it has, and how much ammunition it has

- **gun** variable
- **ammunition** variable

When the player fires (presses their fire key), we need to do the following:

- Check the gun variable , and create the appropriate bullet object
- Decrement the ammunition variable
- Check if the ammunition is less than 0, if so they've run out of bullets and we need to change the state to set the weapon back to the basic weapon, by setting the gun variable




Creating collectable objects is again straight forward – we place objects in the room that, when the tank collides with them, modify the state of the tank in different ways.

- Destroy the object on collision, as it can only be picked up once
- Set the tank's **gun** and **ammunition** variables accordingly

Health works in the same way, an object that, when collided with, resets the tank's health or damage variable. Each tank's health instance variable should obviously be reduced on collision with a bullet, and when it gets to zero the tank should be destroyed.

Finally, we're going to make the different objects **respawn** at different rates in the room, to make things like the big gun and health more scarce and therefore more valuable to the players.

- Make an invisible controller object for each type of collectable object

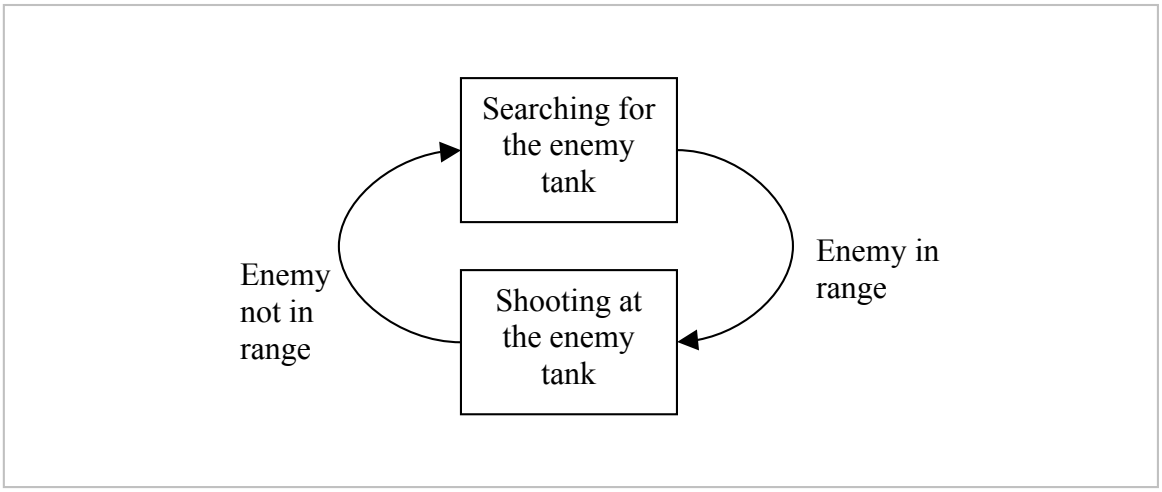
- Use either a **Step Event** and a **Test Chance** action**,** or an **Alarm Event** to create a new instance of the collectable object, either at a random position in the room or at the same position as the invisible controller. If you put it at the same place as the controller, use a **Check Empty** action to see if there's one there already

## "Artificial Intelligence"

Finally, we're going to use a finite state machine to give the second tank some very simple artificial intelligence, so the game can be played with either one or two players. Essentially we're going to create a controller, that on each **Step Event** checks what the tank should be doing.

The finite state machine for the AI tank has two states

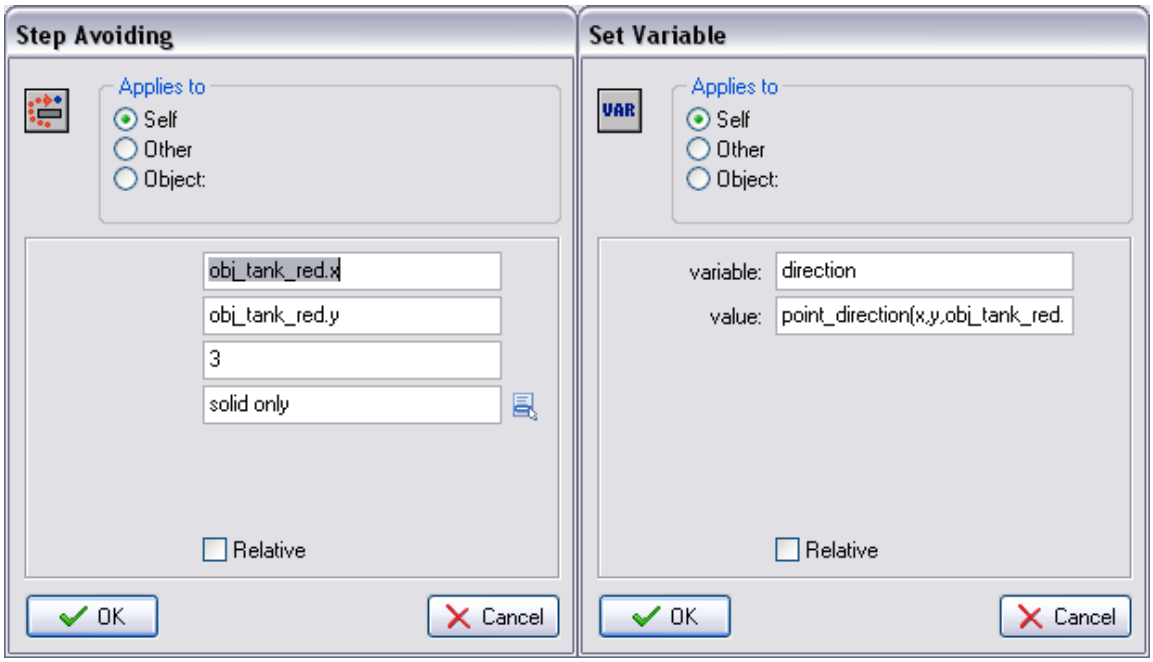
- Searching for the player
- Shooting at the player when they're in range



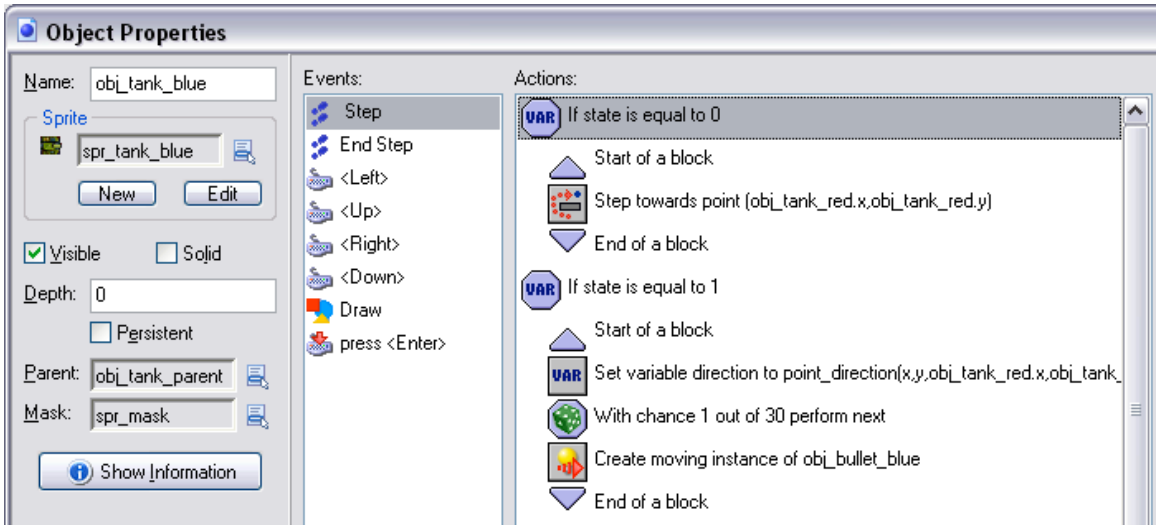

We can break this down into manageable steps by first implementing each of the different states the tank can be in

Create a variable for the three different states, and depending on the value in the **Step Event**, perform a different set of actions

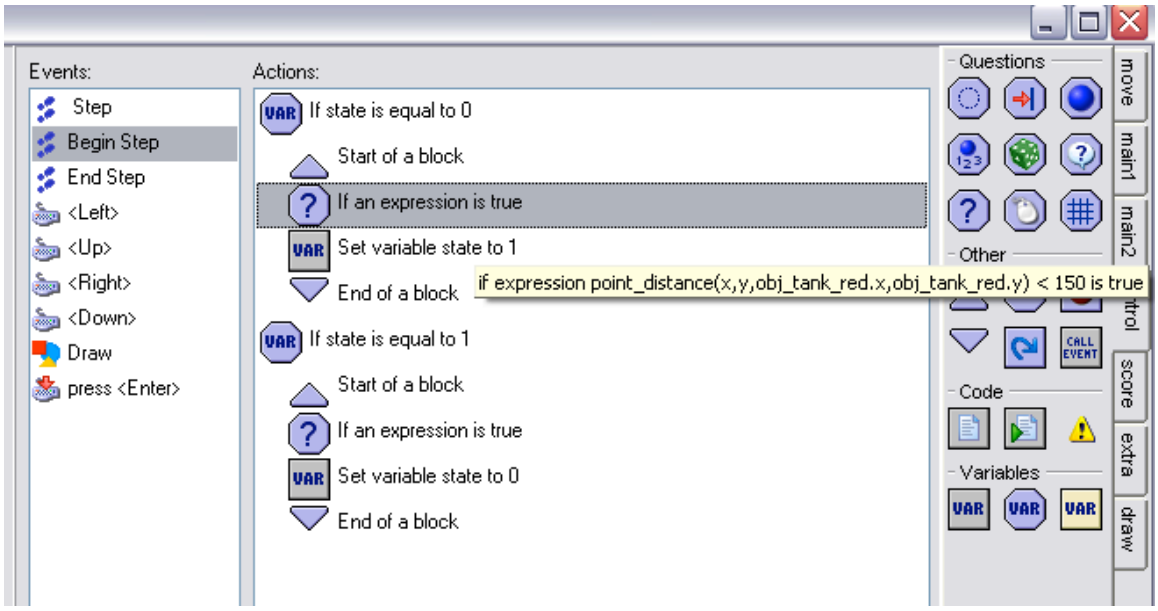
- State = 0 : Searching for the enemy tank
    - In the **Step Event**, use a **Step Avoiding** action to move the tank towards the player's tank
    - If the tank keeps getting stuck on walls, you might need to add a circular **mask sprite** to the tank so that it moves smoothly




- State = 1 : Shooting at the enemy tank
    - Point the tank at the player's tank. We can just set the **direction** variable and the animation and movement of the tank sprite will turn the tank for us. The function **point_direction(x,y,obj_tank_red.x,obj_tank_red.y)** calculates the angle between two points
    - Use a **Test Chance** action to randomly fire bullets at the player's tank




Manually set the state variable in the **Create** event, and make sure both states work as planned. Finally, we need to set up the actions that allow the tank to **transition** between states, so it shoots at the player's tank when it's close enough.




Handle the **Begin Step** event for the tank to check which state the FSM should be in, and to change it accordingly.

- Check how far away the player's tank is using a **Test Expression** action
- The function **point_distance(x1,y1,x2,y2)** calculates the distance between two points
- So, the expression **point_distance(x,y,obj_tank_red.x,obj_tank_red.y) < 150** will be true if the tanks are close together (Why?)
- If the tanks are close enough, then set the **state** variable to **1**, so the tank attacks the player
- When in state 1, check the expression again, so that when the tanks move apart the tank returns to state 0 and chases the player again

If you've got this far, try adding a third state to the tank, where if it gets too damaged it will try to find a health object to pick up before returning to the fight.

You might have noticed that implementing this kind of logic gets complicated using the graphical editor. All of the actions we've used so far have an equivalent that can be called from within Game Maker's built in **scripting language**, which can be either be written as script resources or in snippets within the **Execute Code** action. Have a go at re-implementing the enemy tank logic in a script. You may need to dig out the manual!