CHAPTER 10

Competitive Games: Playing Fair with Tanks

Combat arenas are a popular theme in multiplayer games, because they create extremely compelling gameplay from very simple ingredients. This can often just be an environment filled with weapons that the players can use to wipe each other out. The game that we're going to create in this chapter is exactly that, with futuristic battle tanks. Although games like this are relatively easy to make, care must be taken in their design to ensure that both players feel they are being treated fairly. We'll discuss this more in Chapter 11.

This game will also introduce *views* in Game Maker to help create a larger combat arena. We will also use views to create a split-screen mode, where each player can only see the part of the arena around their own tank.

Designing the Game: Tank War

We're calling this game *Tank War* for obvious reasons. Both players pilot a tank within a large battle arena and the winner is the last one standing. Here's a more detailed description of the game:

Tank War is a futuristic tank combat game for two players. Each player drives his or her own tank through the walled battle arena with the aim of obliterating the other's tank. Once a tank is destroyed, both tanks are respawned at their start position, and a point is awarded to the surviving player. Most walls provide permanent cover, but some can be temporarily demolished to create a way through. There is no ultimate goal to the game, and players simply play until one player concedes defeat.

Each tank has a primary weapon that it can fire indefinitely. Pickups provide a limited amount of ammunition for a secondary weapon, or repair some of the tank's damage:

- *Homing rockets: Always move in the direction of your opponent*
- *Bouncing bombs: Bounce against walls, and can be used to fire around corners*
- *Shields: Are activated to provide a temporary protective shield*
- *Toolbox: Repairs part of the tank's damage*

The game uses a split-screen view divided in two parts (see Figure 10-1). The left part is centered on player one's tank and the right part is centered on player two's tank. There is also a mini-map at the bottom of the screen for locating pickups and the other player.

Player one will move their tank with the A, D, W, and S keys and fire with the spacebar (primary) and Ctrl key (secondary). Player two will control their tank with the arrow keys, and fire with the Enter key (primary) and Delete key (secondary).

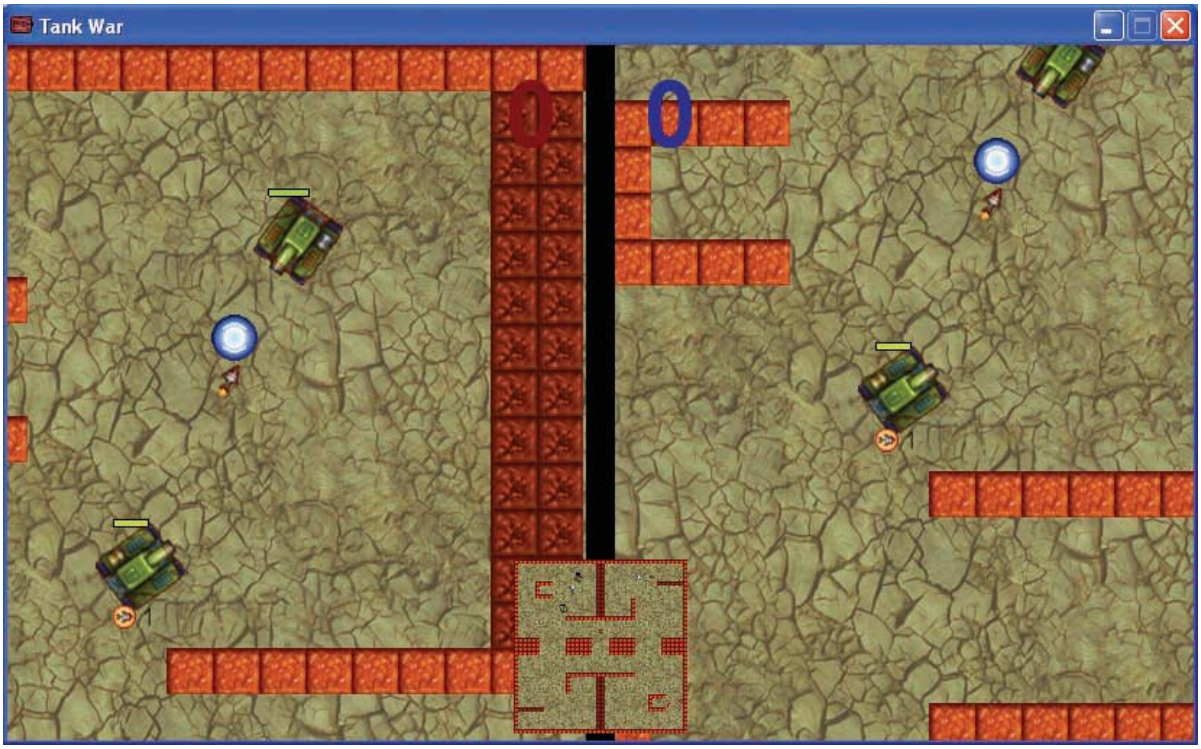


Figure 10-1. *Tank War has a split-screen with a little mini-map at the bottom.*

All resources for this game have already been created for you in the `Resources/Chapter10` folder on the CD.

Playing with Tanks

Our first task is to create the battle arena. This will be a simple environment with two types of walls that will stop tanks and their shells. The first type of wall will be permanent, whereas the second type can be demolished by tank fire but will reappear again after a while.

Creating the arena background and walls:

1. Launch Game Maker and start a new empty game.
2. Create a background resource called `background` using `Background.bmp` from the `Resources/Chapter10` folder on the CD.

3. Create two sprites called `spr_wall1` and `spr_wall2` using `Wall1.gif` and `Wall2.gif`. Disable the **Transparent** property for both sprites.

4. Create a new object called `obj_wall1` and give it the first wall sprite. Enable the **Solid** property and close the object properties. No further behavior is needed.

5. Create a new object called `obj_wall2` and give it the second wall sprite. Enable the **Solid** property and set **Parent** to `obj_wall1`.

Like most of the previous games, this game will have a controller object. For the time being, this will only play the background music but later it will also be responsible for displaying the score.

Creating the controller object and the room:

1. Create a sound resource called `snd_music` using `Music.mp3` from the `Resources/Chapter10` folder on the CD.

2. Create a new object called `obj_controller`, with no sprite. Set **Depth** to `-100` to make sure that the drawing actions we will give it later on are drawn in front of other objects. Add an **Other**, **Game Start** event and include the **Play Sound** action. Set **Sound** to `snd_music` and set **Loop** to **true**.

3. Create a new room and switch to the **settings** tab. Call the room `room_main` and give it an appropriate caption.

4. Switch to the **backgrounds** tab and select the background you created earlier.

5. Switch to the **objects** tab. In the toolbar, set **Snap X** and **Snap Y** to `32`, as this is the size of the wall objects.

6. Create a continuous wall of `obj_wall1` objects around the edge of the room. Also add walls of both types to the interior so that they create obstacles for the tanks (remember that you can hold the Shift key to add multiple instances of an object).

7. Add one instance of the controller object into the room.

Now we'll create our tanks. We'll need different tank objects for each of the two players, but most of their behavior will be identical so we'll create a parent tank object that contains all the common events and actions. In this game we're going to control the tank instances by directly changing their local `direction` and `speed` variables. Remember that the `direction` variable indicates the direction of movement in degrees (0–360 anticlockwise; 0 is horizontally to the right). The `speed` variable indicates the speed of movement in this direction, so a negative value represents a backward movement.

Creating the parent tank object:

1. Create a new object called `obj_tank_parent`, with no sprite.

2. Add a **Create** event and include a **Set Friction** action with **Friction** set to `0.5`. This will cause the tanks to naturally slow down and come to rest when the player is not pressing the acceleration key.

3. Add a **Collision** event with `obj_wall1` and include a **Set Variable** action. Set **Variable** to `speed` and **Value** to `-speed`. This will reverse the tank's movement direction when it collides with a wall.

4. Likewise, add a **Collision** event with `obj_tank_parent` and include a **Set Variable** action. Set **Variable** to `speed` and **Value** to `-speed` (you could also right-click on the previous collision event and select **Duplicate Event** to achieve this).

Creating the two players' tank objects:

1. Create two sprites called `spr_tank1` and `spr_tank2` using `Tank1.gif` and `Tank2.gif`. Set the **Origin** of both sprites to **Center**. Note that these sprites have 60 subimages corresponding to different facing directions for the tanks.

2. Create a new object called `obj_tank1` and give it the first tank sprite. Set **Parent** to `obj_tank_parent` and enable the **Solid** option. Set **Depth** to `-5` to make sure it appears in front of other objects, such as shells, later on.

3. Add a **Keyboard, Letters, A** event and include a **Set Variable** action. Set **Variable** to `direction` and **Value** to `6`, and enable the **Relative** option. This will rotate the tank anticlockwise.

4. Add a **Keyboard, Letters, D** event and include a **Set Variable** action. Set **Variable** to `direction` and **Value** to `-6`, and enable the **Relative** option. This will rotate the tank clockwise.

5. Add a **Keyboard, Letters, W** event and include a **Test Variable** action. Set **Variable** to `speed`, **Value** to `8`, and **Operation** to **smaller than**. Include a **Set Variable** action, setting **Variable** to `speed` and **Value** to `1` and enabling the **Relative** option. This will then only increase the speed if it is smaller than 8.

6. Add a **Keyboard, Letters, S** event and include a **Test Variable** action. Set **Variable** to `speed`, **Value** to `-8`, and **Operation** to **larger than**. Include a **Set Variable** action, setting **Variable** to `speed` and **Value** to `-1` and enabling the **Relative** option. This will only reduce the speed (reverse) if the speed is greater than -8 (full speed backward).

7. Add a **Step, End Step** event. In this event we must set the subimage of the sprite that corresponds to the direction the tank is facing. Include the **Change Sprite** action, setting **Sprite** to `spr_tank1`, **Subimage** to `direction/6` and **Speed** to `0`. As in Galactic Mail, `direction/6` converts the angle the object is facing (between 0 and 360) to the range of images in the sprite (between 0 and 60).

8. We will draw the tank ourselves because later we want to draw more than just the sprite. Add a **Draw** event. Include the **Draw Sprite** action, setting **Sprite** to `spr_tank1` and **Subimage** to `-1` and enabling the **Relative** option.

9. Repeat steps 2–8 (or duplicate `obj_tank1` and edit it) to create `obj_tank2`. This time you should use the arrow key events to control its movement (**Keyboard, Left**, etc.)

10. Reopen the room and put one instance of each tank into it.

Now test the game to make sure everything is working correctly. In case something is wrong, you'll find a version of the game so far in the file `Games/Chapter10/tank1.gm6` on the CD.

Firing Shells

Now the fun begins. In this section we'll create shells for the tanks to shoot at each other, but first we need a mechanism to record the tank's damage and scores. As in Chapter 9, we'll give each tank a variable called `damage` to record the amount of damage it has taken. It will start with a value of 0, and once it reaches 100 the tank is destroyed. We'll also use two global variables called `global.score1` and `global.score2` to record how many kills each tank has made. The controller object will initialize these variables and display their values.

Recording the player's score in the controller object:

1. Create a font called `fnt_score` and select a font like Arial with a **Size** of 48 and the **Bold** option enabled. We only need to use the numerical digits for the score, so you can click the **Digits** button to leave out the other characters in the font. This will save storage space and reduce the size of your `.gm6` and executable game files.

2. Reopen the controller object and select the **Game Start** event. Include a **Set Variable** action with **Variable** set to `global.score1` and **Value** set to `0`. Include another **Set Variable** action with **Variable** set to `global.score2` and **Value** also set to `0`. This creates and initializes the global score variables that will store the player's score.

3. Add a **Draw** event and include a **Set Font** action. Set **Font** to `fnt_score` and **Align** to **right**. Include a **Set Color** action and choose a dark red color.

4. Include a **Draw Variable** action from the **control** tab. Set **Variable** to `global.score1`, **X** to `300`, and **Y** to `10`.

5. Include another **Set Font** action with **Font** set to `fnt_score`, but this time set **Align** to **left**. Include a **Set Color** action and choose a dark blue color.

6. Include a **Draw Variable** action with **Variable** set to `global.score2`, **X** set to `340`, and **Y** set to `10`.

If you run the game now, you should begin with a large 0–0 score displayed on the screen. Next we're going to create two explosions: a large one for when a tank is destroyed, and a small one for when a shell hits something.

Creating the large explosion object:

1. Create a sprite called `spr_explosion_large` using `Explosion_large.gif` and **Center** the **Origin**.

2. Create a sound called `snd_explosion_large` using `Explosion_large.wav`.

3. Create a new object called `obj_explosion_large`. Give it the large explosion sprite and set **Depth** to `-10`. Add a **Create** event and include a **Play Sound** action, with **Sound** set to `snd_explosion_large` and **Loop** set to **false**.

4. Add an **Other, Animation End** event and include the **Restart Room** action.

Creating the small explosion object:

1. Create a sprite called `spr_explosion_small` using `Explosion_small.gif` and **Center** the **Origin**.

2. Create a sound called `snd_explosion_small` using the file `Explosion_small.wav`.

3. Create an object called `obj_explosion_small`. Give it the small explosion sprite and set **Depth** to `-10`. Add a **Create** event and include the **Play Sound** action, with **Sound** set to `snd_explosion_small` and **Loop** set to **false**.

4. Add the **Other, Animation End** event and include the **Destroy Instance** action.

Explosions in hand, we're now ready to create the damage mechanism. The parent tank object will be responsible for initializing the `damage` variable, checking the damage, and drawing the tank's health bar on the screen. It will also be responsible for blowing up the tank when its damage reaches 100, which is why we needed the explosion objects first.

This is all pretty straightforward, and putting this code in the parent tank object will save us some time. However, when the tank blows up we also need to increase the correct player's score—so how do we know which player's tank has died if we are working with the parent object? Fortunately, every instance has a variable called `object_index` that records a number corresponding to the type of object it is. Every object has its own unique number, which can be accessed by using the object name as if it was a variable (in this case `obj_tank1` and `obj_tank2`). So by comparing `object_index` and `obj_tank1` we can tell if the instance is an instance of player one's tank or an instance of player two's.

We'll check the tank's damage in the **Step** event of the parent tank object and increase the appropriate score if it is larger than 100. Then we'll create a large explosion and destroy the tank. The large explosion object will automatically restart the room once the animation is finished.

Adding a damage mechanism to the parent tank object:

1. Reopen `obj_tank_parent` and select the **Create** event. Include a **Set Variable** action with **Variable** set to `damage` and **Value** set to `0`.

2. Add a **Step, Step** event and include a **Test Variable** action. Set **Variable** to `damage`, **Value** to `100`, and **Operation** to **smaller than**. Include an **Exit Event** action so that no further actions are executed if the damage is smaller than 100.

3. Now we need to find out what type of tank we are dealing with. Include a **Test Variable** action with **Variable** set to `object_index`, **Value** set to `obj_tank1`, and **Operation** set to **equal to**. Include a **Set Variable** action with **Variable** set to `global.score2`, **Value** set to `1`, and the **Relative** option enabled. This will then increase player two's score if this instance is player one's tank.

4. Include an **Else** action followed by a **Set Variable** action. Set **Variable** to `global.score1` and **Value** to `1`, and enable the **Relative** option. This will increase player one's score if this instance is player two's tank.

5. Include a **Create Instance** action with **Object** set to `obj_explosion_large` and the **Relative** option enabled.

6. Finally, include a **Destroy Instance** action.

Obviously, we need to draw some kind of health bar so that the players can see how well they are doing. It would be easiest to use the **Draw** event of the parent tank object to do this, but there is a problem. The two tank objects already have their own **Draw** events so they won't normally execute the **Draw** event of the parent object because their own takes priority. Fortunately, we can use the **Call Parent Event** action in the two tanks' own **Draw** events to make sure that the parent's **Draw** event is called as well.

Adding a draw event to the parent tank object to draw the health bars:

1. Add a **Draw** event for the parent tank object.

2. Include a **Set Health** action (**score** tab) and set **Value** to `100-damage`. Damage is the opposite concept to health, so subtracting it from 100 makes this conversion (e.g., 80 percent damage converts to 100 – 80 = 20 percent health).

3. Add a **Draw Health** action. Set **X1** to `-20`, **Y1** to `-35`, **X2** to `20`, and **Y2** to `-30`. Enable the **Relative** option, but leave the other parameters as they are. This will draw a small health bar above the tank. It may seem strange to be using the health functions here as they only work with one health value and we have two players. However, this technique works because we set the health in step 2 using the instance's own `damage` variable, just before we draw the health bar.

4. Reopen `obj_tank1` and select the **Draw** event. Include the **Call Parent Event** action (**control** tab) at the end of the list of actions for this event. This will make sure that the **Draw** event of the parent tank object is also executed.

5. Reopen `obj_tank2` and select the **Draw** event. Include the **Call Parent Event** action (**control** tab) at the end of the list of actions for this event.

With the damage and scoring mechanism in place, we can now create the tank shells. We only want the player's shells to damage their opponent's tank, so we will create a separate shell object for each tank and put common behavior in a shell parent object. We'll also use an alarm clock to give shells a limited life span (and therefore a limited range). Alarm clocks will also help us to temporarily demolish the second wall type when they are hit by shells. We'll move the walls outside the room and use an alarm event to bring them back to their original position after a period of time.

Creating the parent shell object:

1. Create a sprite called `spr_shell` using `Shell.gif` and **Center** the **Origin**. Note that like the tank sprite, this contains 60 images showing the shell pointing in different directions.

2. Create a new object called `obj_shell_parent` and leave it without a sprite (you can set it, but it isn't necessary for the parent as it never appears in the game).

3. Add a **Create** event and include the **Set Alarm** action. Set the **Number of Steps** to `30` and select **Alarm 0**.

4. Add an **Alarm, Alarm 0** event and include the **Destroy Instance** action.

5. Add a **Step, End Step** event and include the **Change Sprite** action. Set **Sprite** to `spr_shell`, **Subimage** to `direction/6`, and **Speed** to `0` (to stop it from animating).

6. Add a **Collision** event with `obj_wall1` and include a **Create Instance** action. Set **Object** to `obj_explosion_small` and enable the **Relative** option. Also include a **Destroy Instance** action to destroy the shell.

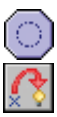
7. Add a **Collision** event with `obj_wall2`. This object must be temporarily removed. Include a **Create Instance** action with **Object** set to `obj_explosion_small` and the **Relative** option enabled. Include a **Jump to Position** action with **X** and **Y** set to `100000`. Also select the **Other** object for **Applies to** so that the wall is moved rather than the shell.

8. Include a **Set Alarm** action and select the **Other** object for **Applies to** so that it sets an alarm for the wall. Select **Alarm 0** and set **Number of Steps** to `300`. Finally, include a **Destroy Instance** action to destroy the shell.

9. Add a **Collision** event with `obj_shell_parent` and include a **Create Instance** action. Set **Object** to `obj_explosion_small` and enable the **Relative** option. Also include a **Destroy Instance** action to destroy the shell.

We now need to make sure that any removed `obj_wall2` instances are returned to their original position when the alarm clock runs out. We will also need to check that the original position is empty first, as we did for the locks in Koalabr8.

Editing the destructible wall object to make it reappear:



1. Reopen the `obj_wall2` object and add an **Alarm, Alarm 0** event. Include a **Check Empty** action with **X** set to `xstart`, **Y** set to `ystart`, and **Objects** set to **All**. Include a **Jump to Start** action.

2. Next include an **Else** action followed by a **Set Alarm** action. Select **Alarm 0** and set **Number of Steps** to `5`. That way, when the position is not empty it will wait five more steps and then try again.

We can now create the actual shell objects.

Creating the players' shell objects:

1. Create a new object called `obj_shell1`. Give it the shell sprite and set its **Parent** to `obj_shell_parent`.

2. Add a **Collision** event with `obj_tank2` and include a **Set Variable** action. Set **Variable** to `damage` and **Value** to `10`, and enable the **Relative** option. Also select the **Other** object for **Applies to** so that the tank's `damage` variable is changed.

3. Include a **Create Instance** action with **Object** set to `obj_explosion_small` and enable the **Relative** option. Also include a **Destroy Instance** action to destroy the shell.

4. Repeat steps 1–3 to create `obj_shell2` using a **Collision** event with `obj_tank1` rather than `obj_tank2`.

Finally, we'll add the actions to make the tanks fire shells. Player one's tanks will shoot shells of type `obj_shell1` when the spacebar is pressed, and player two's tank will shoot shells of type `obj_shell2` when the Enter key is pressed. As in the Wingman Sam game, we'll limit the speed with which the player can fire shells using a `can_shoot` variable. To create bullets that face in the same direction as the tank, we will use the **Create Moving** action and pass in the tank's own `direction` variable.

Adding events to make the tank objects fire shells:

1. Reopen the parent tank object and select the **Create** event. Include a **Set Variable** action with **Variable** set to `can_shoot` and **Value** set to `0`.

2. Select the **Step** event and include a **Set Variable** action at the beginning of the list of actions. Set **Variable** to `can_shoot` and **Value** to `1`, and enable the **Relative** option.

3. Reopen `obj_tank1` and add a **Key Press, <Space>** event. Include the **Test Variable** action, with **Variable** set to `can_shoot`, **Value** set to `0`, and **Operation** set to **smaller than**. Next include the **Exit Event** action so that the remaining actions are only executed when `can_shoot` is larger or equal to 0.

4. Include a **Create Moving** action. Set **Object** to `obj_shell1`, **Speed** to `16`, and **Direction** to `direction`, and enable the **Relative** option. Also include a **Set Variable** action with **Variable** set to `can_shoot` and **Value** set to `-10`.

5. Repeat steps 3–4 for the `obj_tank2`, this time using a **Key Press, <Enter>** event for the key and `obj_shell2` for the **Create Moving** action.

That completes the shells. Test the game carefully and check yours against the one in the file `Games/Chapter10/tank2.gm6` on the CD if you have any problems.

Secondary Weapons

We're going to include secondary weapons and pickups to increase the appeal of the game. Pickups will appear randomly in the battle arena and can be collected by driving into them. Each tank can only have one secondary weapon active at once, so picking up a new weapon

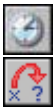
will remove the current one. Toolboxes can also be collected to repair some of the tank's damage, but these will remove any secondary weapons too. All the secondary weapons will have limited ammunition, so the players must take care to make the most of them.

We'll use just one object for all these different kinds of pickups and change its appearance depending on the type of pickup. We'll use a variable called `kind` to record what sort of pickup it is by setting its value to 0, 1, 2, or 3. The value 0 will stand for the homing rocket, 1 for the bouncing bomb, 2 for the shield, and 3 for the toolbox. We can then choose a pickup type at random by using the `choose()` function. To make things more interesting, the pickup will change its `kind` from time to time and jump to a new position. It will also jump to a new position when it is collected by a tank.

Creating the pickup object:

1. Create a sprite called `spr_pickup` using `Pickup.gif`. Note that it consists of four completely different subimages, representing each different kind of pickup.
2. Create a new object called `obj_pickup` and give it the pickup sprite.

3. Add a **Create** event and include the **Set Variable** action. Set **Variable** to `kind` and **Value** to `choose(0,1,2,3)`. This will choose randomly between the numbers in brackets that are separated by commas.



4. Include the **Set Alarm** action for **Alarm 0** and set **Number of Steps** to `100+random(500)`. This will give a random time between 100 and 600 steps or about 3 and 20 seconds. Finally, include a **Jump to Random** action with the default parameters. This will move the instance to a random empty position.

5. Add an **Alarm, Alarm 0** event and include the **Set Variable** action. Set **Variable** to `kind` and **Value** to `choose(0,1,2,3)`.

6. Include the **Set Alarm** action for **Alarm 0** with **Number of Steps** set to `100+random(500)`. Finally, include a **Jump to Random** action.

7. Add a **Collision** event with `obj_tank_parent` and include a **Jump to Random** action.

8. Add a **Draw** event and include the **Draw Sprite** action. Set **Sprite** to `spr_pickup`, **Subimage** to `kind` and enable the **Relative** option.

Now reopen the room and add a few instances of the pickup object to it. Test the game to make sure that the pickups have different images and that they change their type and position from time to time. Also check out what happens when you drive over one with your tank.

We'll also need to record the kind of pickup that has been collected by the tank so that it can change its secondary weapon. We'll use the variable `weapon` for this, where a value of -1 corresponds to no weapon. The variable `ammunition` will indicate how many shots the tank has left of this weapon type. Once `ammunition` reaches 0, `weapon` will be set to -1 to disable the secondary weapon from then on. We'll check the value of the pickup object's `kind` variable in the collision event, and use it to set the tank's `weapon` accordingly.

Editing the parent tank object to record pickups:

1. Reopen `obj_tank_parent` and select the **Create** event.

2. Include a **Set Variable** action with **Variable** set to `weapon` and **Value** set to `-1`. Include a second **Set Variable** action with **Variable** set to `ammunition` and **Value** set to `0`.

3. Add a **Collision** event with `obj_pickup` and include a **Test Variable** action. Set **Variable** to `other.kind`, **Value** to `3`, and **Operation** to **equal to**. A value of 3 corresponds to the toolbox. This needs to repair the tank's damage, so include a **Start Block** action to begin the block of actions that do this.

4. Include a **Set Variable** action with **Variable** set to `weapon` and **Value** set to `-1`. Include a second **Set Variable** action with **Variable** set to `damage` and **Value** set to `max(0,damage-50)`. The function `max` decides which is the largest of the two values you give it (more about functions in Chapter 12). Therefore, this sets the new damage to the largest out of `damage-50` and `0`. In effect, this subtracts 50 from `damage` but makes sure it does not become smaller than 0. Include an **End Block** action.

5. Include an **Else** action, followed by a **Start Block** action to group the actions that are used if this is not a toolbox pickup.

6. Include a **Set Variable** action with **Variable** set to `weapon` and **Value** set to `other.kind`. Include another **Set Variable** action with **Variable** set to `ammunition` and **Value** set to `10`.

7. Finally, include an **End Block** action.

Obviously, it will help players to be able to see the type of secondary weapon they've collected and the ammunition they have remaining for it. We'll display this below each tank using a small image of the pickup. These images have been combined into one sprite again, so we'll need to test the value of `weapon` and draw the corresponding subimage if it is equal to 0, 1, or 2. We can then also draw the value of the variable `ammunition` next to it.

Displaying the secondary weapon in the parent tank object:

1. Create a new sprite called `spr_weapon` using `Weapon.gif`. Note that it consists of three subimages (no image is required for the toolbox).

2. Create a font called `fnt_ammunition` and keep the default settings for it.

3. Select the **Draw** event in `obj_tank_parent` and include a **Test Variable** action. Set **Variable** to `weapon`, **Value** to `-1`, and **Operation** to **larger than**. This will ensure that we only draw something when there is a secondary weapon. Include a **Start Block** action to group the drawing actions.

4. Include the **Draw Sprite** action and select `spr_weapon`. Set **X** to `-20`, **Y** to `25`, and **Subimage** to `weapon`. Also enable the **Relative** option.

5. Include a **Set Color** action and choose black. Then include a **Set Font** action, selecting `fnt_ammunition` and setting **Align** to **left**.

6. Next include a **Draw Variable** action with **Variable** set to `ammunition`, **X** set to `0`, **Y** set to `24`, and the **Relative** option enabled.

7. Finally, include an **End Block** action to conclude the actions that draw the weapon information.

Test the game to check that the weapon icons are displayed correctly when you collect the different weapon pickups. However, so far only the repair kit actually does anything for the player, so let's start by sorting out the rocket. It will behave in much the same way as the shell but automatically starts moving in the direction of the enemy tank. We'll use the same structure of objects as we did for the shell, with common behavior contained in a parent rocket object (`obj_rocket_parent`) and separate rocket objects that home in on the different tanks (`obj_rocket1` and `obj_rocket2`). We'll also make `obj_shell_parent` the parent of `obj_rocket_parent` so that it inherits `obj_shell_parent`'s **Collision** and **Alarm** events. However, we don't want `obj_rocket_parent` to have the same **Create** and **End Step** events as `obj_shell_parent` so we'll give it new versions of these events that give the rocket a longer lifetime and draw the correct sprite.

Creating the parent rocket object:

1. Create a sprite called `spr_rocket` using `Rocket.gif` and **Center** the **Origin**.
2. Create a new object called `obj_rocket_parent` and set **Parent** to `obj_shell_parent`.

3. Add a **Create** event and include the **Set Alarm** action. Set **Number of Steps** to `60` and select **Alarm 0**.

4. Add a **Step, End Step** event and include a **Change Sprite** action. Select the rocket sprite, then set **Subimage** to `direction/6` and **Speed** to `0`.

Next we create the two actual rocket objects.

Creating the actual rocket objects:

1. Create a new object called it `obj_rocket1` and give it the rocket sprite. Set **Parent** to `obj_rocket_parent`.

2. Add a **Create** event and include the **Move Towards** action. Set **X** to `obj_tank2.x`, **Y** to `obj_tank2.y`, and **Speed** to `8`.

3. Add a **Collision** event with `obj_tank2` and include a **Set Variable** action. Select **Other** from **Applies to** (the tank), set **Variable** to `damage` and **Value** to `10`, and enable the **Relative** option.

4. Include a **Create Instance** action, selecting `obj_explosion_small` and enabling the **Relative** option. Also include a **Destroy Instance** action.

5. Create `obj_rocket2` in the same way, but move toward `obj_tank1` in the **Create** event, and add a **Collision** event with `obj_tank1` for the actions in steps 3 and 4.

Finally, we need to make it possible for the tanks to fire rockets. We'll check whether they have the weapon and ammo in the **Key Press** event of the sceondary fire key. If they do, then we'll create the rocket and decrease the ammunition. When it reaches 0, we'll set `weapon` to –1 to disable it.

Adding events to shoot rockets for the tank object:

1. Reopen the first tank object and add a **Key Press, <Ctrl>** event. Include a **Test Variable** action, with **Variable** set to `can_shoot`, **Value** set to `0`, and **Operation** set to **smaller than**. Next include the **Exit Event** action so that the remaining actions are only executed when `can_shoot` is larger than or equal to 0.

2. Include the **Test Variable** action, with **Variable** set to `weapon`, **Value** set to `0`, and **Operation** set to **equal to**. Next include a **Test Instance Count** action with **Object** set to `obj_tank2`, **Number** set to `0` and **Operation** set to **larger than**. Follow this with a **Create Instance** action for `obj_rocket1`, and enable the **Relative** option. This creates a rocket only when it is the current secondary weapon and the other tank exists (this avoids a rare error when the other tank has just been destroyed).

3. Next we need to decrease the ammunition. Include a **Set Variable** action with **Variable** set to `ammunition`, **Value** set to `-1`, and the **Relative** option enabled. Include a **Test Variable** action with **Variable** set to `ammunition`, **Value** set to `1`, and **Operation** set to **smaller than**. Follow this with a **Set Variable** action with **Variable** set to `weapon` and **Value** set to `-1`.

4. Finally, include a **Set Variable** action with **Variable** set to `can_shoot` and **Value** set to `-10`.

5. Repeat steps 1–4 for `obj_tank2`, using a **Key Press, Others, <Delete>** event and creating `obj_rocket2`.

Now we'll create the bouncing bomb secondary weapon in a similar fashion. It behaves in the same way as the shell except that it bounces against walls.

Creating the bouncing bomb objects:

1. Create a sprite called `spr_bouncing` using `Bouncing.gif` and **Center** the **Origin**.

2. Create a new object called `obj_bouncing_parent` and set its **Parent** to `obj_shell_parent`.

3. Add a **Collision** event with `obj_wall1` and include the **Bounce** action. Select **precisely** and set **Against** to **solid objects**.

4. Add a similar **Collision** event with `obj_wall2`.

5. Add a **Step, End Step** event and include a **Change Sprite** action. Select `spr_bouncing`, set **Subimage** to `direction/6`, and set **Speed** to `0`.

6. Create a new object called `obj_bouncing1` and give it the bouncing bomb sprite. Set its **Parent** to `obj_bouncing_parent`.

7. Add a **Collision** event with `obj_tank2` and include a **Set Variable** action. Select **Other** from **Applies to**, set **Variable** to `damage`, set **Value** to `10`, and enable the **Relative** option. Include a **Create Instance** action for `obj_explosion_small` and enable the **Relative** option.

8. Include a **Destroy Instance** action.

9. Repeat steps 6 and 7 to create `obj_bouncing2` using a **Collision** event with `obj_tank1`.

Before we add actions to make the tank objects shoot bouncing bombs, we'll create the final special weapon: the shield. This is a bit more complicated as it allows the player to temporarily make their tank invincible. Activating the shield will set a new variable called `shield` to 40, and display a shield sprite. The value of `shield` will be reduced by 1 in each step until it falls below 0 and the shield is disabled again. We'll check the value of `shield` each time the tank is hit and only increase its damage when `shield` is less than 0.

Editing the parent tank object to support shields:

1. Create sprites called `spr_shield1` and `spr_shield2` using `Shield1.gif` and `Shield2.gif` and **Center** their **Origins**.

2. Reopen the parent tank object and select the **Create** event. Include a **Set Variable** action with **Variable** set to `shield` and **Value** set to `0`.

3. Select the **Step** event and include a **Set Variable** action at the start of the list. Set **Variable** to `shield`, set **Value** to `-1`, and enable the **Relative** option.

4. Reopen `obj_shell1` and select the **Collision** event with `obj_tank2`. Include a **Test Variable** action directly above the **Set Variable** that increases the damage. Select **Other** from **Applies to**, then set **Variable** to `shield`, **Value** to `0`, and **Operation** to **smaller than**. Now the damage will only be increased when the tank has no shield.

5. Repeat step 4 for objects `obj_shell2`, `obj_rocket1`, `obj_rocket2`, `obj_bouncing1`, and `obj_bounding2`.

6. Reopen `obj_tank1` and select the **Draw** event. Include a **Test Variable** action at the start of the action list. Set **Variable** to `shield`, **Value** to `0`, and **Operation** to **larger than**. Follow this with a **Draw Sprite** action for `spr_shield1` with the **Relative** option enabled.

7. Repeat step 6 for `obj_tank2`, this time drawing `spr_shield2`.

Now all that remains is to adapt the tanks so that both the bouncing bombs and the shields can be used.

Editing tank objects to shoot bombs and use shields:

1. Reopen `obj_tank1` and select the **Key Press, <Ctrl>** event.

2. Include a **Test Variable** action below the **Create Instance** action that creates `obj_rocket1`. Set **Variable** to `weapon`, **Value** to `1`, and **Operation** to **equal to**. Follow this with a **Create Moving** action for `obj_bouncing1`, setting **Speed** to `16` and **Direction** to `direction`, and enabling the **Relative** option.

3. Include another **Test Variable** action below this, with **Variable** set to `weapon`, **Value** set to `2`, and **Operation** set to **equal to**. Follow this with by a **Set Variable** action with **Variable** set to `shield` and **Value** set to `40`.

4. Repeat steps 1–3 for `obj_tank2`, adapting the **Key Press, <Delete>** event and creating `obj_bouncing2`.

This completes all the secondary weapons and the game should now be fully playable. We encourage you to play it a lot with your friends, to make sure everything is working as it should. You'll find the current version on the CD in the file `Games/Chapter10/tank3.gm6`.

Views

Currently, our playing area is quite small and both players can see all of it at once. However, we can create more interesting gameplay by giving each player a limited "window" into a much larger playing area. This can easily be achieved in Game Maker using *views*. We'll use two views to create a split screen, in which the left half of the screen shows the area around the first tank and the right half shows the area around the second tank. Later we use a third view to display a little mini-map as well.

To understand the concept of views, you need to appreciate that there is a distinction between a room and the window that provides a view of that room on the screen. Up to now, rooms have always been the same size as the window and the window has always showed the entire contents of the room. However, rooms can be any size you like, and views can be used to indicate the specific area of the room that should appear in the window. We're going to create a room that's twice the width of a normal room with an equal height (see Figure 10-2). The green rectangle shows the size of a normal room, and the red and blue squares show the size of the views we will give to each player in the room. To create these views, we will need to specify the following information on the **views** tab in the room properties:

- **View in room**: This is an area of the room that needs to be displayed in the view. The **X** and **Y** positions define the top-left corner of this area and **W** and **H** specify the width and height of it.

- **Port on screen**: This is the position on the window where the view should be shown. The **X** and **Y** positions define the top-left corner of this area and **W** and **H** specify the width and height of it. If the width and height are different from the size of the view area, then the view will be automatically scaled to fit. Game Maker will also automatically adapt the size of the window so that all ports fit into it.

- **Object following**: Specifying an object here will make the view track that object as it moves around the room. **Hbor** and **Vbor** specify the size of the horizontal and vertical borders that you want to keep around the object. The view will not move until the edge of the screen is closer than this distance from the object. Setting **Hbor** to half the width of the view and **Vbor** to half the height of the view will therefore maintain the object in the center. Finally, **Hsp** and **Vsp** allow you to limit the speed with which the view moves (–1 means no limit).

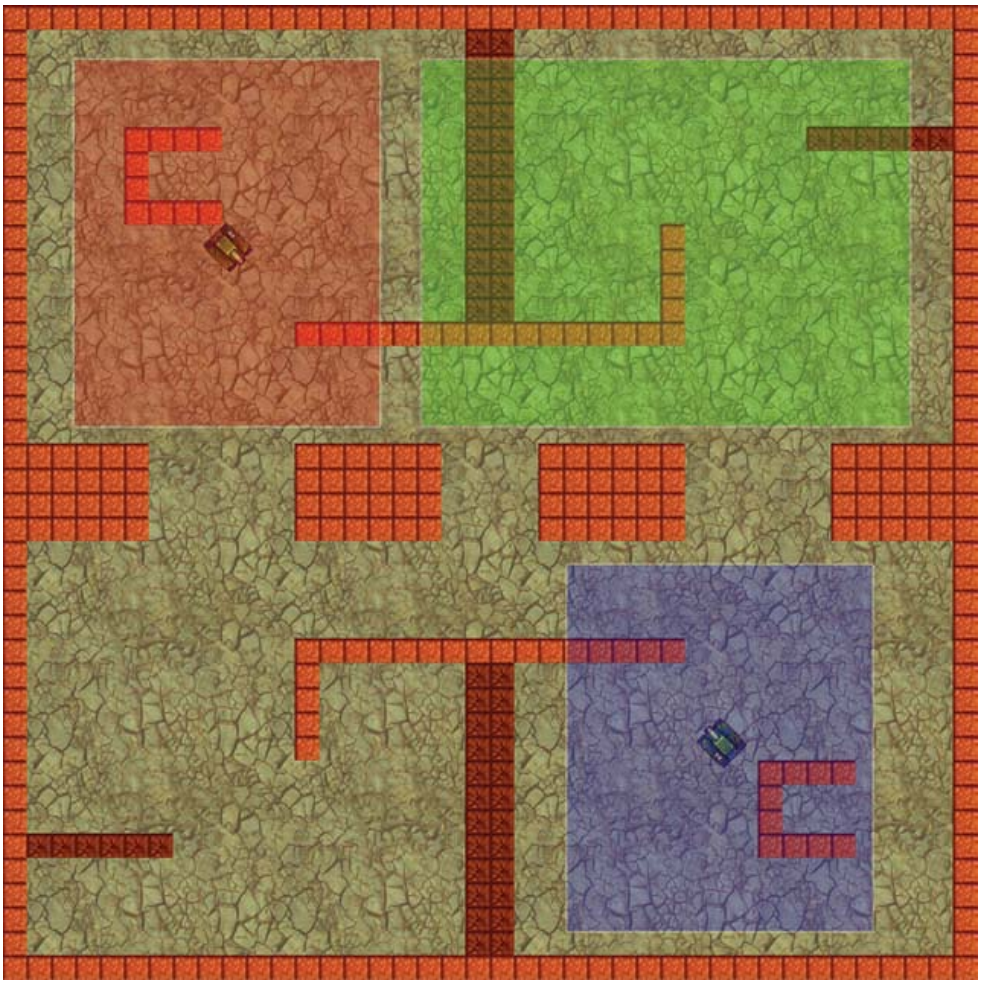


Figure 10-2. *We'll create a large room, much bigger than a normal window (green rectangle), and provide views into it for each of the tanks (red and blue squares).*

You can specify up to eight different views, but you'll probably only need one or two. Let's adapt our game's room to use two views.

Editing the room resource to provide two views:

1. Reopen the main room and switch to the **settings** tab.
2. Set both the **Width** and **Height** of the room to `1280`, to create a much larger room.
3. Switch to the **objects** tab and add wall instances to incorporate the extra playing area. Start the tanks close to two opposite corners and add six pickup instances. Also don't forget that the room needs exactly one instance of the controller object.
4. Switch to the **views** tab and select the **Enable the use of Views** option. This activates the use of views in this room.
5. Make sure that **View 0** is selected in the list and enable the **Visible when room starts** option. We will use this view for player one.

6. Under **View in room** set **X** to `0`, Y to `0`, W to `400`, and H to `480`. The **X** and **Y** positions of the views don't really matter in this case as we will make them follow the tanks. Nonetheless, notice that lines appear in the room to indicate the size and position of the view.

7. Under **Port on screen** set **X** to `0`, **Y** to `0`, **W** to `400`, and **H** to `480`. This port will show player one's view on the left side of the screen.

8. Under **Object following** select `obj_tank1`, then set **Hbor** to `200` and **Vbor** to `240`. The form should now look like Figure 10-3.

9. Now select **View 1** in the list and enable the **Visible when room starts** option. We will use this view for player two.

10. Under **View in room** set **X** to `0`, **Y** to `0`, **W** to `400`, and **H** to `480`.

11. Under **Port on screen** set **X** to `420`, **Y** to `0`, W to `400`, and H to `480`. This places the second view to the right of the first view with a little space between them.

12. Under **Object following** select `obj_tank2`, and set **Hbor** to `200` and **Vbor** to `240`.

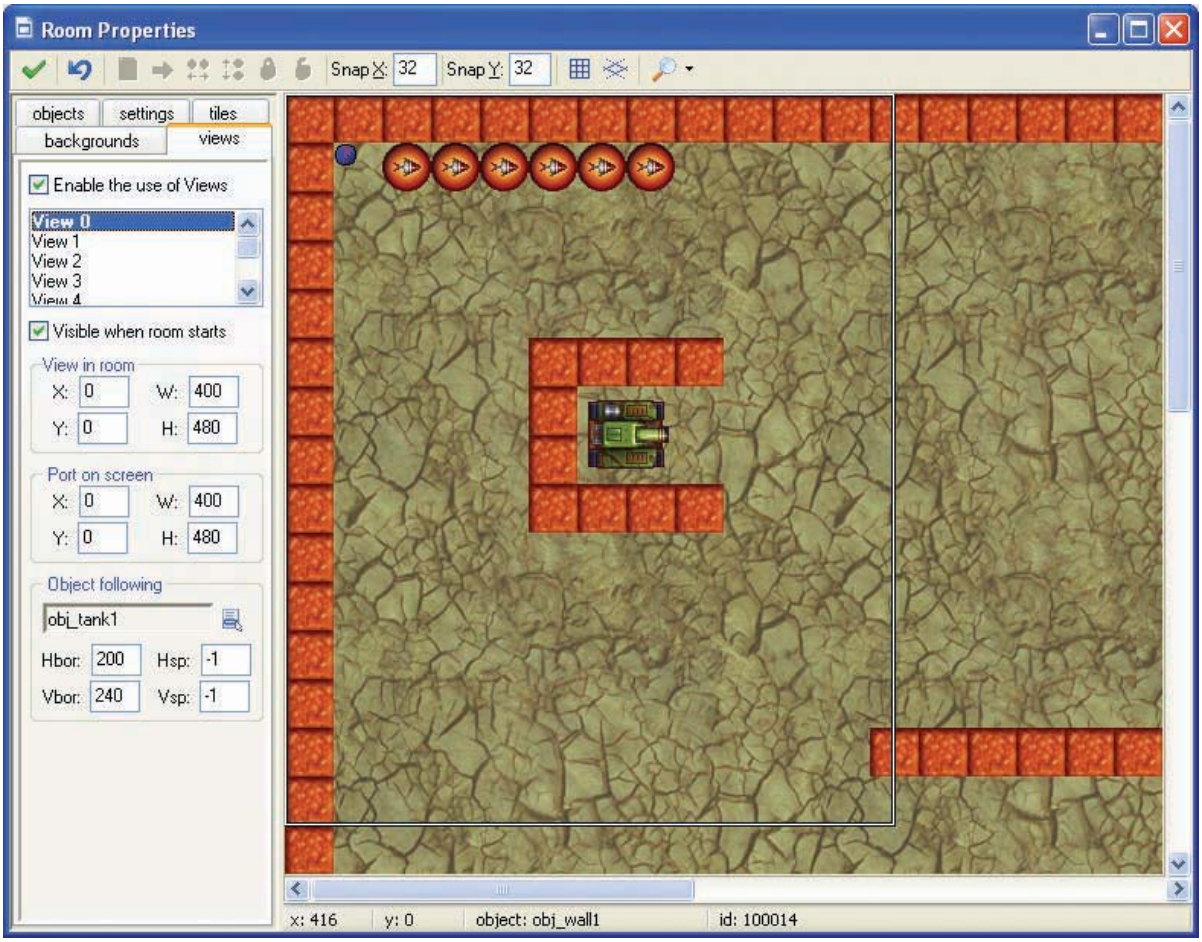


Figure 10-3. *This is how the form should look when the values for View 0 have been set.*

And that's it. Easy, wasn't it? Run the game and you should be able to play in the new split-screen mode.

■ **Tip** The empty region between the views defaults to the color black. You can change this in the **Global Game Settings** on the **graphics** tab under **Color outside the room region**.

Have you noticed something strange? The score is displayed at a fixed position in the room so you can only see it if you drive up to it! To fix this we need to draw it at a changing position relative to the player's views. The score for player one needs to appear in the top-right corner of View 0 and the score for player two needs to appear in the top-left corner of View 1. Game Maker provides variables that we can use to obtain the positions of views. `view_xview[0]` and `view_yview[0]` indicate the current x- and y-positions of View 0 while `view_xview[1]` and `view_yview[1]` indicate the x- and y-positions of View 1.

Unfortunately, this does not solve the problem completely. To explain why, you'll need to understand what Game Maker is doing when you use views. For each view, Game Maker draws the whole room, including all the backgrounds, objects, and **Draw** events; clips the visible area to the size of the view; and then copies it to the required position on the window. This means the **Draw** event of the controller object (that draws the score) is called twice, once for drawing each of the views. So, to display the score in the correct place we need to know which view is currently being drawn. Game Maker allows us to check this using the variable `view_current`, which will be 0 for View 0 and 1 for View 1. Therefore, we can test the value of this variable in the **Draw** event of the controller object and draw the score of the appropriate tank relative to the position of the current view.

Editing the controller object to draw the score relative to the view position:

1. Reopen the controller object and select the **Draw** event.

2. Include a **Test Variable** action before the **Draw Variable** action that draws the score for player one. Set **Variable** to `view_current`, **Value** to `0`, and **Operation** to **equal to**.

3. Edit the **Draw Variable** action that draws player one's score. Change **X** to `view_xview[0]+380` and **Y** to `view_yview[0]+10`.

4. Include a **Test Variable** action before the **Draw Variable** action that draws the score for player two. Set **Variable** to `view_current`, **Value** to `1`, and **Operation** to **equal to**.

5. Edit the **Draw Variable** action for player two. Change **X** to `view_xview[1]+20` and **Y** to `view_yview[1]+10`. The action list should now look like Figure 10-4.

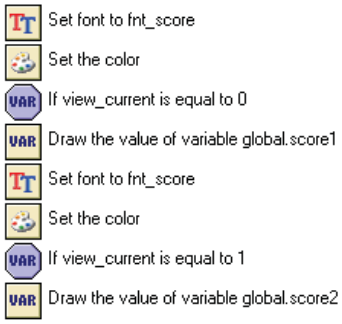


Figure 10-4. *These actions draw the scores correctly for each view.*

Run the game to check that the score is displayed correctly.

We'll now add a little mini-map to help the player see where they are. This mini-map shows the entire room, so that both players can see the location of their opponents and the pickups in the room. Creating a mini-map is very simple using views, as we can create an additional view that includes the whole room but scales it down to a small port on the screen.

Adding a view to create a mini-map:

1. Reopen the main room and switch to the **views** tab.
2. Select **View 2** in the list and enable the **Visible when room starts** option.
3. Under **View in room** set **X** to `0`, **Y** to `0`, **W** to `1280`, and **H** to `1280` (the entire room).
4. Under **Port on screen** set **X** to `350`, **Y** to `355`, **W** to `120`, and **H** to `120`. No object needs to be followed.

And that finishes the game for this chapter. Run it and check that it all works. There are a few final improvements you might want to make. You should add some **Game Information** and you might want to change some of the **Global Game Settings**. For example, you might not want to display the cursor but might want to start in full-screen mode or add a loading image of your own for the game.

Tip To improve the mini-map and make it more "iconic," you could make the different objects draw something different when the variable `view_current` is equal to 2. For example, the pickup object could simply display a red disk and the walls could draw black squares.

Congratulations

That's another one complete! We hope you enjoyed making this game and playing it with your friends. The final version can be found on the CD in the file `Games/Chapter10/tank4.gm6`. You encountered some important new features of Game Maker in this chapter, including views, which can be used to create all sorts of different games.

There are many ways in which you could make Tank War more interesting. You could create different arenas for the players to compete in. Some could be wide and open while others could have close passageways. You could also add other types of walls, perhaps stopping shells but not the tanks, or even the other way around. You could create muddy areas that reduce your speed, or slippery areas that make it difficult to steer your tank. Of course, you can also add other types of secondary weapons, such as guns that fire sideways or in many different directions. You could even drop mines or create holes in the ground. You could also add a front-end to the game, displaying the title graphic that is supplied. You're the designer and it's up to you.

We'll be staying with our Tank War example in the next chapter as we explore the game design issues involved in creating multiplayer games. We've got some different versions of the game for you to play and you'll be balancing tanks, so you'd better go and find some king-sized scales!