



Introduction to Programming

Pass Task 2.3: Circle Moving

Overview

Using control flow it is possible to create programs that respond to user input. In this task you will create a small program that allows the user to move a circle around on the screen.

Purpose: Learn to use the control flow statements within a program to respond to user actions.

Task: Create a program that allows the user to move a circle around the screen.

Time: This task should be completed before the start of week 6.

Resources:

- Chapter 5 of the Programming Arcana
- Swinburne CodeCasts ([YouTube Channel](#), [iTunesU](#))
 - [Control Flow](#)
 - [Branching with if statements](#)
 - [Repeating code with loops](#)
- Syntax Videos
 - [Repeat](#), [Compound Statement](#), [If Statement](#), [While](#), [Case](#)

Submission Details

You must submit the following files to Doubtfire:

- Circle Moving program source code.
- Screenshot of the program in action.

Make sure that your task has the following in your submission:

- The program must move the circle, and ensure it remains on the screen.
- Code must follow the Pascal coding convention used in the unit (layout, and use of case).
- The code must compile and the screenshot show it working.

Instructions

So far all of the *SwinGame* programs we have built contained a sequence of actions (such as shape drawing) controlled by delays. However, this situation does not allow the user to interact with the program. For example, the program is ended by a 'hard coded' delay. What if the user wanted to look at the drawing for longer than specified in our delay, or for less time than in our delay? What we need is for the window remains open **until** the user requests it to be closed.

Another common feature that a user expects in programs like games is to be able to move things around. For example, **if** the user presses the move right (►) arrow on the keyboard **then** the character moves to the right. In this task you will create a program that uses control flow mechanisms (such as repeat/until and if/then) to allow the program to respond to user input (such as close window requests, keyboard actions, and mouse actions).

To do this we will be creating an **event loop** which calls procedures to recognise and respond to user input (such as closing windows when requested and moving characters using the keyboard). Table 1 shows some of *SwinGame*'s functions and procedures for working with input.

Table 1: Procedures to assist processing of user input in SwinGame

<i>Function / Procedure</i>	<i>Does</i>
ProcessEvents ();	Listens for user input. No input can be received unless this is called.
MouseX() : Single;	X location of the mouse. A MouseY() function also exists.
MouseClicked(button) : Boolean;	Was the mouse button clicked? Buttons are LeftButton and RightButton
WindowCloseRequested() : Boolean;	Has the user asked to close the Window?
KeyDown(key) : Boolean;	Is the key currently held down? Keys are in the format ...Key eg AKey, LeftKey, UpKey, SpaceKey
KeyTyped(key) : Boolean;	Was the key typed? (Pressed then released)

Hint: The navigation arrows on the keyboard are identified in *SwinGame* by UpKey, DownKey, LeftKey and RightKey. So you can check for the user typing the 'up' arrow key by using KeyTyped(UpKey)

1. Download the **SwinGame Pascal - Project Template.zip** file from Blackboard.
2. Extract the zip file to your code directory (e.g. Documents/Code)
3. Rename the **Project Template** folder to **CharacterMoving**
4. Open a **Terminal** window and navigate to your *CharacterMoving* directory.
5. Write the code to implement a basic *SwinGame* program called *CharacterMoving* using the following code.

```
program CharacterMoving;
uses SwinGame, sgTypes;

procedure Main();
begin
    OpenGraphicsWindow('Character Moving', 800, 600);
    Delay(5000);
end;

begin
    Main();
end.
```

- You need to use both *SwinGame* and *sgTypes* to get access to key codes

Note: We are adding another library to the **uses** specification in the program, **sgTypes**. This will give us access to the key codes and mouse buttons.

6. Notice the window closes after 5 seconds, as the program's instructions end. To correct this we can add an **event loop**. Implement the following pseudocode in your **Main**.
 - The event loop will be located in **Main**, and will loop until the user closes the window.
 - **Process Events** needs to be called *once* each event loop to update *SwinGame* with the actions that have occurred since the last time through the loop.

Procedure: **Main**

Steps:

- 1: **Open** a **Graphics Window** with title 'Character Moving' that is 800x600
- 2: Repeat
- 3: **Process Events**
- 4: Until **Window Close** is **Requested**

7. Switch back to the Terminal and compile and run the program. It should now remain open until you close the window.

The event loop repeats code over and over as the program runs. This now means that you can create an interactive program. Inside this loop you can listen for user *events* and then update the program's data.

8. Alter Main to draw a circle on the screen, using variables for the circle's x and y location.

```
-----  
Procedure: Main  
-----  
Local Variables:  
- x, y: Single data for the circle's location  
Steps:  
1: Open a Graphics Window with title 'Character Moving'  
   that is 800x600  
2: Assign x the value 400  
3: Assign y the value 300  
4: Repeat  
5:     Process Events  
6:     Clear the Screen to ColorWhite  
7:     Fill a Circle using ColorGreen, at location x,y  
       with a radius 110  
8:     Refresh the Screen limiting it to 60 FPS  
9: Until Window Close is Requested
```

Note: To ensure a consistent game speed you can use `RefreshScreen(60)` to limit the refresh rate to 60 frames per second (FPS)

9. Switch to the Terminal, compile and run the program. You should be able to see a green circle in the centre of the screen.
- The x and y variables in Main store all of the data for this game: the location of the circle. As these are the only variables in the "game" (which is being run by the steps in the Main procedure), these are the only things that can change.

The next step will involve using **if** statements to selectively run sections of your code. This can be used to ensure the computer only runs certain code when a condition is met. For example, we could only move the character to the left when the left arrow key is held down.

10. Alter Main to update the x variable when the user is holding down. This will...

- Make x smaller if the user is holding down the left arrow key, which will move the circle left
- Make x larger if the user is holding down the right arrow key, which will move the circle right

```

-----
Procedure: Main
-----

Local Variables:
- x, y: Single data for the circle's location
Steps:
1: Open a Graphics Window with title 'Character Moving'
   that is 800x600
2: Assign x the value 400
3: Assign y the value 300
4: Repeat
5:     Process Events

6:     if the LeftKey Key is Down then
7:         Assign x, the value x - 1
8:     if the RightKey Key is Down then
9:         Assign x, the value x + 1

10:    Clear the Screen to ColorWhite
11:    Fill a Circle using ColorGreen, at location x,y
       with a radius 110
12:    Refresh the Screen limiting it to 60 FPS
13: Until Window Close is Requested

```

Note: Control flow in this pseudocode is grouped based in **indentation**. Pay attention when there is more than one statement within the control flow statement, as this is likely to require a compound statement to group (i.e., *begin ... end* in Pascal)

11. Switch back to the Terminal and compile and run the program. You should be able to move the circle using the left and right arrow keys.
12. Add code to use **UpKey** and **DownKey** to move the circle along the Y axis (up and down the screen).
13. Switch back to the Terminal and compile and run the program. You should be able to move the circle left, right, up, and down using the arrow keys.

14. If you keep your finger on the one arrow key long enough, the circle will disappear off the edge of the screen.
15. Before addressing this problem, let's fix the use of the literal value 110 as the radius. Instead, add a constant named **CIRCLE_RADIUS** and set it to 110.

Tip: Avoid having "magic numbers" in your code. Numbers like 110 do not have as much meaning as a constant like **CIRCLE_RADIUS**. Later when you read or change the code these constants will make it easier to understand and change.

16. Alter Main to use the **CIRCLE_RADIUS** constant when it draws the circle.
17. Now, adjust the program to ensure that the circle remains on the screen.

Tip: To find the width of the current screen you can call the *SwinGame* function **ScreenWidth()**. **ScreenHeight()** will give you the height of the screen.

Hint: When moving right you should only change the x value *if* the key is held down *and* x is less than **ScreenWidth() - the circle's radius**. Eg:

```
if KeyDown(RightKey) and (x + CIRCLE_RADIUS < ScreenWidth()) then ...
```

Hint: The left edge of the screen is at X 0.

18. Switch to the Terminal. Compile and run the program, and test that you cannot move the circle off the screen to the left or right.
19. Now, change the radius of the circle to be 150. Compile and run the program again, and if you have coded it correctly the circle should still be able to go right up to the edge of the screen.

Once your program is working correctly you can prepare it for your portfolio.