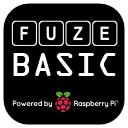
Hello once again and welcome to our FUZE BASIC tutorial. To further our exploration into the exciting world BASIC we’re going get things moving. Over the next few pages we’re adding moving enemies, well big rocks actually, and the ability to fire. Of course firing a bullet is one thing but what happens when it hits something? Intrigued? Read on…

Last month we added our player ship graphic and some movement commands to get us started. The movement commands need to be explained as there is some very useful stuff going on.  
  
Generally the method for checking if a key has been pressed is to use the **INKEY** command as this will return the key being pressed when you check. For example “**Key=INKEY**” will store the value -1 in the variable “Key” if no key is pressed or the ASCII value of any key that is being pressed. The Space Bar has an ASCII value of 32, the “A” key is 65 and the “a” is 97 and so on.

You can write a simple program to help you determine the **INKEY** values;

Before we get started though please get back to the FUZE BASIC environment using the icon on the Desktop. Then go to the Editor **(F2)** and enter this little example.

|  |
| --- |
| **The Editor** |
| **CYCLE**  **PRINT INKEY**  **REPEAT** |

**RUN (F3)** this and hold down different keys to see what value they give you. Notice however that if you hold down two keys simultaneously only one of the numbers will be displayed. This means we can’t read two or more keys being pressed at the same time. This is not good for games programming so we added a command to FUZE BASIC do exactly that.

To demonstrate this let’s first get back to business. Press **ESC** to exit the program and then **F2** twice to return to direct mode so we can load our program back in;

**Dir (press Enter)**

You should see a Directory called MagPi

Type;

**CD MagPi (press Enter)**

**Load MagPi (press Enter)**

Press **F2** to go to the Editor. Hopefully your program will be as we left it but if not you might want to go through and make sure all is as it should be. The full program listing is as follows;

|  |
| --- |
| **The Editor** |
| **// MagPi Game**  **PROC Setup  CYCLE  PROC CheckControls**  **PROC ScreenUpdate**  **REPEAT  END**  **DEF PROC CheckControls**  **UpKey = scanKeyboard (scanUp)**  **DownKey = scanKeyboard (scanDown)**  **LeftKey = scanKeyboard (scanLeft)**  **RightKey = scanKeyboard (scanRight)**  **IF UpKey THEN ShipY = ShipY + 1**  **IF DownKey THEN ShipY = ShipY - 1**  **IF LeftKey THEN ShipX = ShipX - 1**  **IF RightKey THEN ShipX = ShipX + 1**  **ENDPROC**  **DEF PROC ScreenUpdate**  **plotSprite (Ship, ShipX, ShipY, 0)**  **UPDATE**  **ENDPROC**  **DEF PROC Setup**  **HGR**  **updateMode = 0**  **ShipX = 0**  **ShipY = gHeight / 2**  **Ship = newSprite (1)**  **loadSprite ("Player2.bmp", Ship, 0)**  **setSpriteTrans (Ship, 255, 0, 255)**  **ENDPROC** |

If you **RUN (F3)** the program you can see we can use the cursor keys to move the ship around the screen. This is handled using the **scanKeyboard(scanxxxx)** command. The “RightKey = **scanKeyboard (scanRight)**” checks to see if the Up cursor key is being pressed, and if it is stores a 1 in the variable “RightKey” or a 0 if it has not been pressed. This can be applied to every key so it’s easy to check for more than one key press with a simple **IF**, **AND**, **THEN** statement.  
  
Press the **ESC** key and then **F2** to return to the editor.

We need to add a few restrictions so the player cannot zoom off the screen. At the same time we’re going to add a simple animation to the player’s ship. Firstly, modify just the **DEF PROC CheckControls** section so it is as follows;

|  |
| --- |
| **The Editor – In PROC CheckControls** |
| **DEF PROC CheckControls**  **ShipID = 1**  **UpKey = scanKeyboard (scanUp)**  **DownKey = scanKeyboard (scanDown)**  **LeftKey = scanKeyboard (scanLeft)**  **RightKey = scanKeyboard (scanRight)**  **IF UpKey AND ShipY <= (gHeight - getSpriteH (Ship)) THEN**  **ShipY = ShipY + 4**  **ShipID = 2**  **ENDIF**  **IF DownKey AND ShipY >= 0 THEN**  **ShipY = ShipY - 4**  **ShipID = 0**  **ENDIF**  **IF LeftKey AND ShipX >= 0 THEN ShipX = ShipX - 4**  **IF RightKey AND ShipX <= gWidth / 2 THEN ShipX = ShipX + 2**  **ENDPROC** |

Don’t RUN it at this point as you will just get an error.

The **IF** statements check to see if the **ShipX** or **ShipY** positions are off the screen and if so will not allow further movement. The check “**IF UpKey AND ShipY <= (gHeight** - **getSpriteH (Ship))** “ looks much more complicated than it is. **gHeight** is a system variable that contains the maximum height of the current screen mode and **getSpriteH** checks the height of the sprite specified. We don’t want the sprite to be allowed to go any lower than the bottom but this must take the height of the sprite into consideration. Try removing this part of the check to see what happens.

The **ShipID** variable is going to contain the player Sprite ID so we can display different sprites just by changing this setting. You can see from the above the default setting is 1 and then depending if the up or down keys are pressed it can be 0 or 2. Hmm.. One graphic for up, one for down and a default one for when neither up or down are pressed. The mind boggles, although probably not!

For the sake of gameplay we’ve also increased the speed dramatically as the ship now moves four pixels at a time, unless it is moving forward where it has to work a bit harder at only 2 pixels.

Because we’ve introduced new variable, we need to change other parts too. Add this Procedure right after the **END** statement.

|  |
| --- |
| **The Editor – Add after END statement** |
| END  **DEF PROC DrawShip**  **plotSprite (Ship, ShipX, ShipY, ShipID)**  **ENDPROC** |

As we’re about to have a lot more going on we’ll be using separate functions for each of the main sprites. This one specifically draws the player’s ship at the X and Y position. Change the beginning of the program to;

|  |
| --- |
| **The Editor** |
| **// MagPi Game**  **PROC Setup**  **CYCLE**  **PROC CheckControls**  **PROC DrawShip**  **UPDATE**  **REPEAT**  **END** |

The **UPDATE** statement is now issued from the main loop. This will keep things running smoothly as everything will be updated at once. Because of this we no longer need the **ScreenUpdate** Procedure so this can be removed;

|  |
| --- |
| **The Editor – Remove DEF PROC ScreenUpdate** |
| ~~DEF PROC ScreenUpdate~~  ~~plotSprite (Ship, ShipX, ShipY, 0)~~  ~~UPDATE~~  ~~ENDPROC~~ |

Nearly there.. To introduce a new variable and add the extra ship graphics update the **DEF PROC Setup** section to read;

|  |
| --- |
| **The Editor – In PROC Setup** |
| **DEF PROC Setup**  **HGR**  **updateMode = 0**  **ShipX = 0**  **ShipY = gHeight / 2**  **Ship = newSprite (3)**  **loadSprite ("Player1.bmp", Ship, 0)**  **loadSprite ("Player2.bmp", Ship, 1)**  **loadSprite ("Player3.bmp", Ship, 2)**  **setSpriteTrans (Ship, 255, 0, 255)**  **ShipID = 0**  **ENDPROC** |

As you can see we have added two extra sprites. The clever part is that only one main sprite container is required to hold all the sprites for that graphic. We then use an index to determine which one to display. The 0, 1 and 2 at the end of each **loadSprite** command determines the index value. Then when we want to display the sprite we use **plotSprite(name, x, y, index).** You can have many sprites indexed so complex animations can be achieved by simply changing the index.

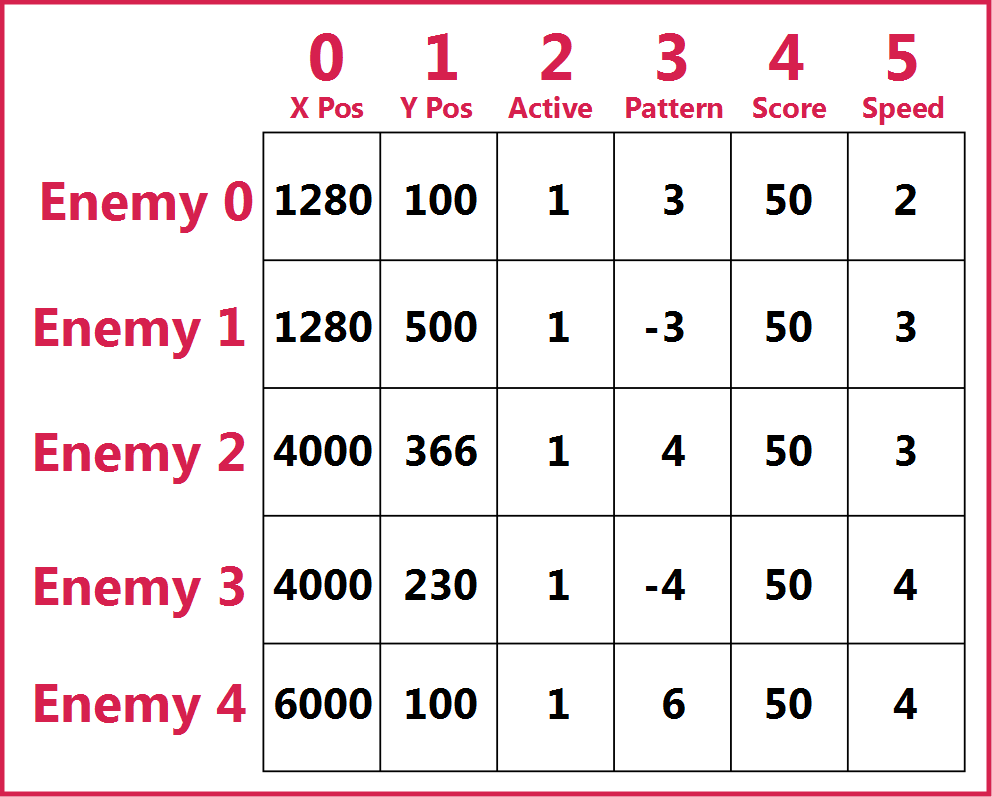
Ok, now you can **RUN (F3)** the program. If you get any errors you’ll need to go back and debug. Just make sure everything matches the listing here and it should be fine. Now when you move around with the cursor keys things should be much faster and best of all the ship will change depending if it’s going up or down. It’s a very simple but effective method.

So, what next?

It’s time to add some scary monsters. Actually monsters would take a bit more space to cover in this article so we’re going with rocks but they are big and ugly ones if that helps.

There is, I’m afraid, no way out of this next part. We need to add a large section to the Setup procedure to introduce our enemies. We could have kept things very simple and just gone for one enemy sliding across the screen at a time but where’s the fun in that? We’re going for waves of sixteen (split into two lots of eight) at a time at different speeds and with changing flying patterns. The price to pay is in the typing so, heads down and get on with it.

|  |  |
| --- | --- |
| **The Editor – In PROC Setup** | **Description** |
| DEF PROC Setup  HGR  updateMode = 0  ShipX = 0  ShipY = gHeight / 2  Ship = newSprite (3)  loadSprite ("Player1.bmp", Ship, 0)  loadSprite ("Player2.bmp", Ship, 1)  loadSprite ("Player3.bmp", Ship, 2)  setSpriteTrans (Ship, 255, 0, 255)  ShipID = 0  **EnemyMax = 63**  **eID = 0**  **DIM Enemy(EnemyMax, 6)**  **EnemyID = 0**  **EnemyX = 0**  **EnemyY = 0**  **EnemyActive = 1**  **EnemyVariation = 0**  **EnemyScore = 0**  **EnemySpeed = 0**  **DIM Rock(EnemyMax)**  **FOR num = 0 TO EnemyMax CYCLE**  **Rock(num) = newSprite (1)**  **loadSprite ("BigRock.bmp", Rock(num), 0)**  **setSpriteTrans (Rock(num), 255, 0, 255)**  **REPEAT**  **EnemyCount = 0**  **UNTIL EnemyCount > EnemyMax CYCLE**  **READ EnemyX**  **READ EnemyY**  **READ EnemyVariation**  **READ EnemyScore**  **READ EnemySpeed**  **EnemyScore = EnemyScore \* EnemySpeed**  **DATA 1280, 100, 3, 50, 2**  **DATA 1280, 500, -3, 50, 2**  **DATA 4000, 366, 4, 50, 3**  **DATA 4000, 230, -4, 50, 3**  **DATA 6000, 100, 6, 50, 3**  **DATA 6000, 500, -6, 50, 3**  **DATA 11000, 400, 5, 50, 4**  **DATA 11000, 300, -5, 50, 4**  **FOR num = 0 TO 7 CYCLE**  **Enemy(EnemyCount + num, 0) = Rock(EnemyCount + num)**  **Enemy(EnemyCount + num, 1) = EnemyX + num \* getSpriteW (Rock(0))**  **Enemy(EnemyCount + num, 2) = EnemyY**  **Enemy(EnemyCount + num, 3) = EnemyActive**  **Enemy(EnemyCount + num, 4) = EnemyVariation**  **Enemy(EnemyCount + num, 5) = EnemyScore**  **Enemy(EnemyCount + num, 6) = EnemySpeed**  **REPEAT**  **EnemyCount = EnemyCount + 8**  **REPEAT**  **ENDPROC** | The grey part is what you should already have so just start entering from **EnemyMax=63**  The next few lines introduce lots of new variables.  Most important is the **DIM Enemy(EnemyMax,6)** statement as this is a very powerful type of variable called an array. It allows us to store multiple bits of information in an index rather than just a single number in a single variable.  See the graph paper picture for a more detailed explanation.  Now we need to fill the **Enemy()** and **Rock()** arrays with information.  First off is the main Sprite ID using the **newSprite** command.  The rest are the X & Y coordinates, the pattern used, the score value and the speed of the rock.  We’re using **READ** and **DATA** commands to easily add a whole load of information in one go.  The score is based on 50 \* the speed but we’re not including that this month so you’ll have to be patient.  The rocks are configured in waves of eight.  The main loop reads each of the lines of data and stores them in individual variables.  This smaller FOR LOOP is filling in all the information in blocks of eight at a time. |

Again, don’t RUN yet as it won’t work without a few more changes.

**Array’s**  
The array variable is very similar to using a database where you have a record and then various bits of information against that record. We have a Record ID, say “1” and then store information within this like, name, address, post code and so on. In the case of an array variable we can then refer to any part of this database with a simple index command.

Looking at the picture can you tell me the value stored in our Enemy array at position 3,3?

Of course you can, it’s 7… just kidding, it is of course “-4”

Using arrays allows us to store massive amounts of information all instantly accessible with a simple index command. For example “Speed = Enemy(4,5)” should now make sense.

Notice, and as usual with computer programming, everything starts counting from zero.

Back to our program;

Once again if we **RUN** at this point nothing will happen as we still need to add a few more bits elsewhere first.

Firstly this adds a call to a new procedure called **DrawEnemy**;

|  |
| --- |
| **The Editor** |
| **// MagPi Game**  **PROC Setup**  **CYCLE**  **PROC CheckControls**  **PROC DrawShip**  **PROC DrawEnemy**  **UPDATE**  **REPEAT**  **END** |

  
And this adds the actual procedure. Once again add this right after the **END** statement;

|  |
| --- |
| **The Editor – Add after END statement** |
| **DEF PROC DrawEnemy**  **FOR eID = 0 TO EnemyMax CYCLE**  **IF Enemy(eID, 3) THEN**  **Enemy(eID, 1) = Enemy(eID, 1) - Enemy(eID, 6)**  **EY = Enemy(eID, 2) + COS (Enemy(eID, 1)) \* Enemy(eID, 4) \* 10**  **plotSprite (Enemy(eID, 0), Enemy(eID, 1), EY, 0)**  **ENDIF**  **REPEAT**  **ENDPROC** |

**RUN** to see how things are looking. All going well you should now have wave after wave of very scary rocks flying towards you across the screen. Press **ESC** to stop the program when you’ve seen enough.

The **PROC DrawEnemy** section looks really complicated but when broken down it is, as always, easier than is first apparent;

|  |  |
| --- | --- |
| **DEF PROC DrawEnemy**  **FOR eID = 0 TO EnemyMax CYCLE**  **IF Enemy(eID, 3) THEN**  **Enemy(eID, 1) = Enemy(eID, 1) - Enemy(eID, 6)**    **EY = Enemy(eID, 2) + COS (Enemy(eID, 1)) \* Enemy(eID, 4) \* 10**  **plotSprite (Enemy(eID, 0), Enemy(eID, 1), EY, 0)**  **ENDIF**  **REPEAT**  **ENDPROC** | When the procedure is called it sets up a **FOR LOOP** to go through the entire 64 enemy sprites (EnemyMax is set to 63 and as we start counting from 0 this makes 64 in total)  The first **IF** statement checks if the Enemy is active or not. If it is **THEN**  Reduce the **X** Position **(Enemy(eID,1)** by the speed stored in **Enemy(eID,6).**  I don’t like this bit as it makes me sound clever and I’m not, but here goes. We are working out the Y position using the cosine of the X position. This causes the Y position to go up and down. The (eID,4) is gives us a variable to adjust the strength of the wave (the height).  As the X position is always moving to the left a simple wave motion is formed.  If you actually are clever then you can work out ways to make very complex patterns. Simple waves are about as much as I can manage! |



How are we doing so far?

One last thing to do this month and that’s to add the promised fire power. We are going to keep it simple as once again it would take too much space to do anything really fancy but this will hopefully give you something to work with.

Ok here we go;

At the beginning add the **PROC DrawBullet** statement;

|  |
| --- |
| **The Editor** |
| **// MagPi Game**  **PROC Setup**  **CYCLE**  **PROC CheckControls**  **PROC DrawShip**  **PROC DrawEnemy**  **PROC DrawBullet**  **UPDATE**  **REPEAT**  **END** |

And;

|  |
| --- |
| **The Editor – In PROC CheckControls** |
| LeftKey = scanKeyboard (scanLeft)  RightKey = scanKeyboard (scanRight)  **SpaceKey = scanKeyboard (scanSpace)**  **IF SpaceKey AND NOT Fire THEN PROC Bullet**  IF UpKey AND ShipY <= (gHeight - getSpriteH (Ship)) THEN  ShipY = ShipY + 4  ShipID = 2  ENDIF |

Then this at the end of the **DEF PROC Setup** procedure;

|  |
| --- |
| **The Editor – In PROC Setup** |
| EnemyCount = EnemyCount + 8  REPEAT  **DIM Shot(3)**  **Shot(0) = newSprite (1)**  **loadSprite ("Bullet.bmp", Shot(0), 0)**  **setSpriteTrans (Shot(0), 255, 0, 255)**  **Fire = 0**  ENDPROC |

Then add these two Procedures directly below the **END** statement. These two display the bullet and work out if it has hit anything with the **spriteCollidePP (Shot(0),2)** command. More on this in a minute, but for now get busy and enter this lot;

|  |
| --- |
| **The Editor – Add after END statement** |
| **DEF PROC DrawBullet**  **IF Shot(1) > gWidth THEN**  **hideSprite (Shot(0))**  **Shot(3) = 0**  **Fire = 0**  **ENDIF**  **IF Shot(3) THEN**  **Shot(1) = Shot(1) + 6**  **plotSprite (Shot(0), Shot(1), Shot(2), 0)**  **Hit = spriteCollidePP (Shot(0), 2)**  **IF Hit > 0 AND Hit <= 64 THEN**  The **PROC Bullet** procedure is very straightforward as it simply works out where the bullet should appear based on the current position of the player ship. The **PROC DrawBullet** procedure has a lot more going on and introduces a few new sprite commands;  Firstly if the bullet goes off the screen then set it to inactive and hide the sprite with the **hideSprite(Shot(0))** command.  **Hit = spriteCollidePP (Shot(0), 2)** checks the sprite to see if it is in contact with any other sprite and if so stores that sprite ID into the variable Hit. We then check to see if it is a Rock and if so hide the rock, the bullet and set them both to inactive so we don’t draw them again elsewhere. We also reset the Fire variable so we can shoot again.  There are two kinds of collision detection; **spriteCollidePP(ID, accuracy)** and **spriteCollide(ID). PP** stands for **Pixel Perfect** so very accurate collisions can be checked for while the standard version just checks the sprite’s bounding box.  **Enemy(Hit - 1, 3) = 0**  **hideSprite (Hit)**  **hideSprite (Shot(0))**  **Shot(3) = 0**  **Fire = 0**  **ENDIF**  **ENDIF**  **ENDPROC**  **DEF PROC Bullet**  **Fire = 1**  **Shot(1) = ShipX + getSpriteW (Ship) + 8**  **Shot(2) = ShipY + getSpriteH (Ship) / 2 - 10**  **Shot(3) = 1**  **ENDPROC** |

Ok that’s all we’ve got space for this month, actually I might be in trouble for taking up so much already ☹

Next issue will see the final part in this series. I hope to wrap things up with a few more collisions, scoring and Start and Game Over scenes to tie up any loose ends.

If you haven’t already noticed we are, in association with the very nice people at The MagPi, running a competition for the best game entry submitted using FUZE BASIC on a Raspberry Pi. Check last month’s issue for details. You can download the FUZE BASIC Boot image from the resource page on the [www.fuze.co.uk](http://www.fuze.co.uk) website.