Bubblewrap Popper

Introduction

You don't have to be an expert in DarkBASIC Pro to create the type of program that is available commercially. In fact, it's surprising just how little effort some programs require.

In this article we are going to create a Bubblewrap Popping program. For a true addict with plenty of time on his hands there is nothing more satisfying than popping those little dimples on a sheet of bubblewrap! Of course, in this day and age we have to think very carefully before indulging in such pleasures: we are after all destroying a recyclable resource which has caused much pollution of the planet in its creation. There's also the danger that we will do irreparable damage to our hearing from the constant popping noise of the bubbles - with true addicts even holding the wrap up to an ear for a greater appreciation of that satisfying noise. And of course, bubblewrap is never to be found when your desire is at its peak!

But here is the solution, a bubblewrap popping program that uses up none of the planet's precious resources (I'm assuming, of course, that you have your own wind turbine or solar panels for recharging your laptop's batteries)!

The Plan

We will develop the program in two parts. The first of these will contain only a single bubble. This will demonstrate the basic concepts of how the program operates. The second program will have a 10 x 10 sheet of bubbles so that a satisfying number of bubbles can be popped before the program terminates.

The Resources

Before we think of programming, we need to think out what resources in the way of images and sound files were going to need for this project. Only four items are required:

- An image of an unpopped bubble
- An image of a popped bubble
- An image of a single black pixel
- A sound file containing a popping noise

The requirement for the black image will become clear later.

Creating the Resources

If you are creating the program solely for your own use, then you are free to download the required resources from the internet. A quick "google" will locate everything you need.

On the other hand, if you intend to let others use your program - particularly if you intend to make financial gain from the product - then you need to get permission to use anything you find on the internet. Even images and sounds belong to their creators and you are violating copyright laws when you use them without permission.

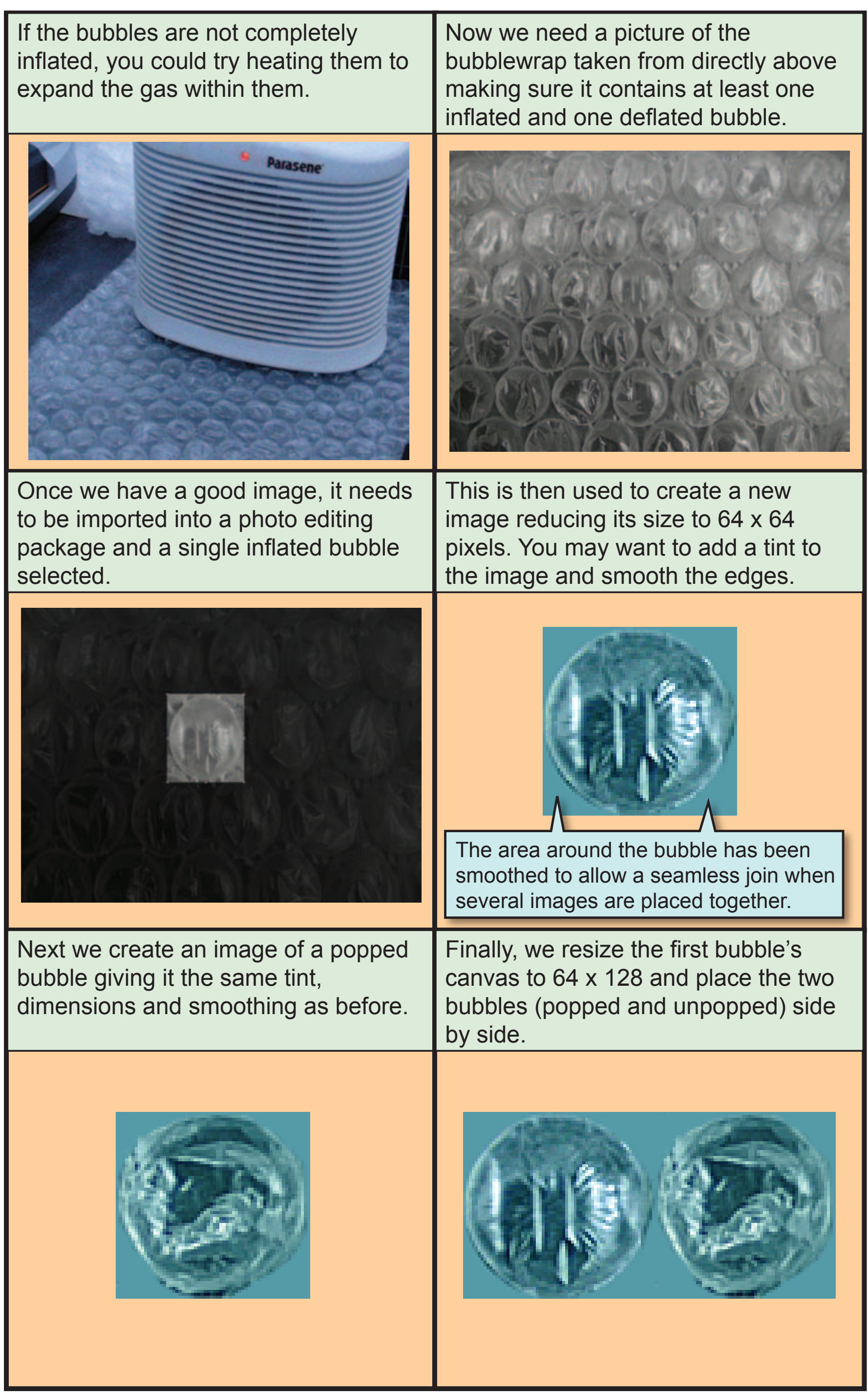
For this article I am going to show you how to create your own images and sound.

The Images

First we need to get hold a some bubblewrap and a camera to create our initial images. The steps involved are shown in FIG-1.1 below.

FIG-1.1

Creating the Bubble Image



This final image is then saved as *bubble.bmp*.

We still need one more image containing a single black pixel. The reason for this image will be come clear later.

Activity 1.1

Create the 64 x 128 image of the bubbles as described in FIG-1.1 saving the file as *bubble.bmp*.
Using your image editing program, create an image containing a single black pixel saving the file as *black.bmp*.

(If you don't have time to do this, you can download the images from our website on the DarkBASIC Downloads page.)

The Sound

To capture the sound, all we need is a microphone, a piece of bubblewrap and sound-editing software.

If you don't have sound editing software, you can download Power Sound Editor for free from *www.free-sound-editor.com*.

If your computer doesn't have a built in microphone, you'll need to plug the microphone into the soundcard. The remaining steps in the process are shown in FIG-1.2.

FIG-1.2

Creating the Sound File

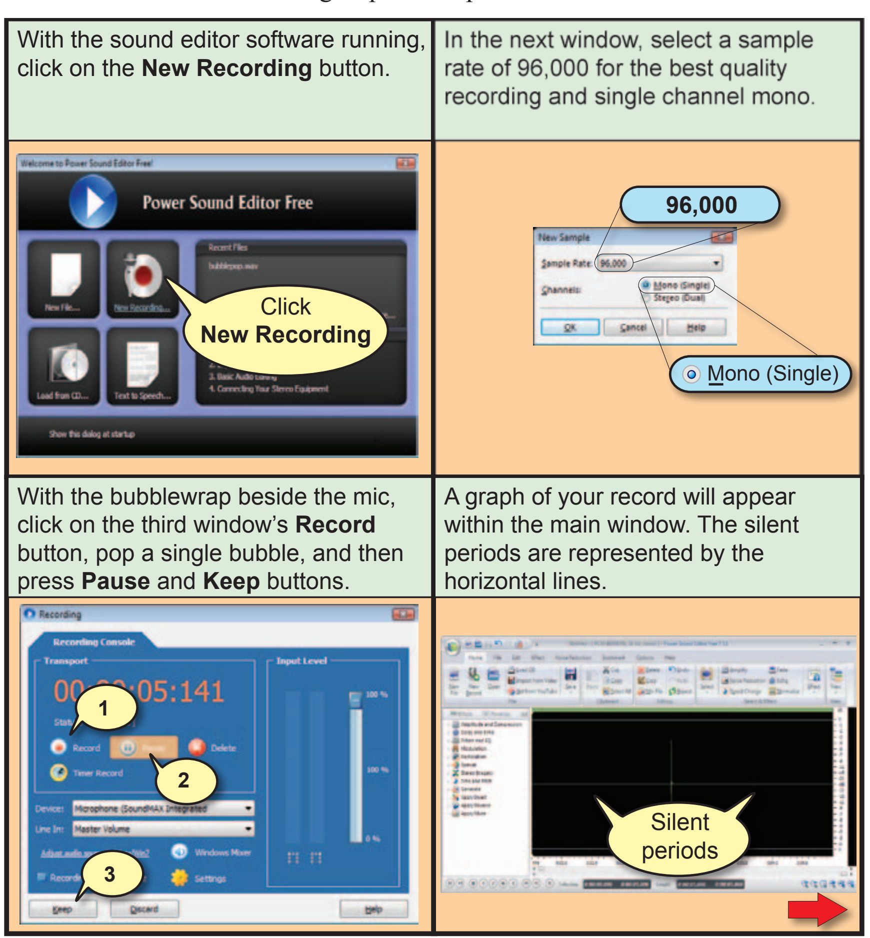
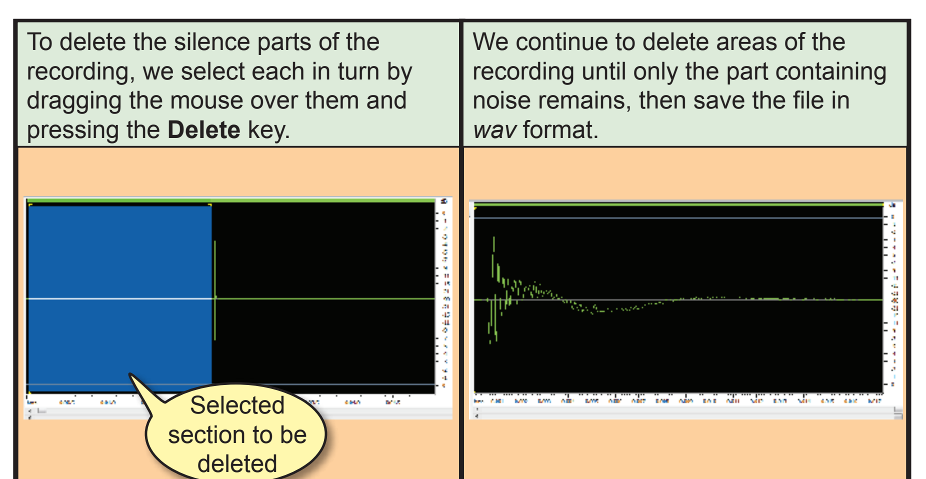


FIG-1.2
(continued)

Creating the Sound File



Activity 1.2

Create a sound file using the methods described in FIG-1.2. Save the file as *pop.wav.*

(If you don't have time to do this, you can download the sound file from our website on the DarkBASIC Downloads page.)

The Program

We are going to create two versions of the program. The first will have only a single bubble and is being created just to make sure that the basic popping mechanism works correctly. The second version will have a 10 x 10 sheet of bubbles to allow for a more satisfying number of pops.

The Logic

Before we get started on any coding we need to have a clear idea of the logic involved in the program we are about to write and should take the trouble to write this down using structured English. This little bit of extra effort will help clarify in our own heads what is required of the program and help us produce a complete, bug-free result in the shortest time.

The first version of the program requires the following logic:

```
Set up the screen resolution
Load up the sound file
Create a "black" sprite
Create the "bubble" sprite showing unpopped bubble
Set the status of the "bubble" sprite to "unpopped"
REPEAT
    Position the black sprite under the mouse pointer
    IF mouse over bubble AND bubble unpopped AND left mouse button pressed
    THEN
        Play pop sound
        Change "bubble" sprite to show popped bubble
        Set status of the "bubble" sprite to "popped"
    ENDIF
UNTIL "bubble" sprite status is "popped"
```

Don't worry if not all of the logic is clear to you, we'll describe each part in more detail as we implement it.

Implementation

Set up the screen resolution

An LCD screen may be set to various resolutions but for the best image you should always set the resolution to match the number of pixels specified for the screen. My own screen has a resolution of 1920 x 1200, so that's the resolution I'm using here. You should change this line to match your own screen's resolution. In addition, you want to set the number of bits defined for each pixel. This determines the number of colours that can be shown. Most modern screens allow 32 bits per pixel. So to set the screen resolution I use the line:

```
SET DISPLAY MODE 1920, 1200,32
```

Load the sound file

Before we can play a sound within a program, we need to load the file containing it and assign the file a sound file number. The statement in this case is:

```
LOAD SOUND "pop.wav",1
```

The file name string should contain drive and folder details if the file is not in the same folder as the DBPro program we are creating. Note that the sound file has been assigned the number 1. This number will be used to refer to the file from this point on.

Create the "black" sprite

This sprite is to contain our single black pixel image. To create a sprite we must first load the image to be used in the sprite and then use that image to create the sprite. An image is loaded in much the same way as a sound file: we need to specify the file name and assign it a numeric value. This is done with the line:

```
LOAD IMAGE "black.bmp",1
```

Notice that the image has been assigned the numeric identity of 1. The fact that both the sound file and the image have both been assigned the value 1 does not present a problem since they are different types of objects. It would, however, be invalid to assign two files of the same type identical numbers.

With the image loaded, we can now create the "black" sprite. The SPRITE command must be supplied with several values. These specify the number to be assigned to the sprite, the position of the sprite on the screen and the ID of the image used to create the sprite. We are going to give the sprite an ID of 1, position it on the screen at coordinates 640,480 and use image 1 in its creation so the code required is:

```
SPRITE 1,640,480,1
```

Create the “bubble” sprite showing the unpopped bubble

The bubble sprite is somewhat different from the black sprite in as much as it needs to contain two parts or frames. The first frame shows the “unpopped” bubble and the second frame shows the “popped” bubble. This type of sprite is known as an animated sprite (although in this case it is not being used to simulate movement). The statement required to create an animated sprite is different from that required for a standard sprite such as the “black” sprite that contains only one image.

There is no need to create a separate statement for loading the picture being used into an image as we did before. Instead, the sprite and image are created in a single statement which specifies the number being assigned to the sprite, the name of the image file, the number of columns and rows the image is to be split into and the number to be assigned to the image. The statement required here is:

```
CREATE ANIMATED SPRITE 2,"bubble.bmp",2,1,2
```

This specifies that the sprite has the ID 2, the file being loaded is called *bubble.bmp* and that it is to be split into 2 columns and 1 row; the image created as a by-product is to be assigned the ID of 2.

Set the status of the “bubble” sprite to “unpopped”

The program needs to know the state of the bubble so that it does not allow a “popped” bubble to be popped for a second time. This is done by creating a variable and assigning it one value while the “bubble” sprite is showing the unpopped bubble and a second value when the popped bubble is showing. We will use the value 1 to represent the unpopped state and 2 to represent the popped state. So our code it will be:

```
bubblestatus = 1
```

Position the black sprite under the mouse pointer

The black sprite needs to be positioned beside the tip of the mouse pointer. To do this we need to find the *x* and *y* coordinates of the mouse pointer. This is achieved using MOUSEX() and MOUSEY() statements which return the mouse’s *x* coordinate and the *y* coordinate respectively. The sprite is positioned using the SPRITE statement we used earlier when positioning the bubble. The exact command this time is:

```
SPRITE 1, MOUSEX(),MOUSEY(),1
```

Now at last, we can reveal why the “black” sprite is needed! This sprite exists to make it easier for us to detect when the mouse is positioned over the bubble sprite. There is no command in DarkBASIC Pro for directly detecting when the mouse is over a sprite, but it does have a command for detecting when one sprite is positioned over another. So, by having the “black” sprite always positioned under the mouse pointer, we can easily detect when they are overlapping. The sprite is black because the colour black is normally invisible within a sprite meaning that the “black” sprite is hidden from the user.

IF mouse over bubble AND bubble unpopped AND left mouse button pressed

The IF statement contains three conditions that must be tested. We can check if the mouse is over the bubble by checking to see if the “black” sprite (sprite 1) is in collision with the bubble sprite (sprite 2). This is done using the expression:

```
SPRITE COLLISION(1,2) = 1
```

The SPRITE COLLISION statement returns the value 1 if the two sprites specified within the parentheses are overlapping (it returns zero if they are not).

The bubble is unpopped if the bubble status is set to 1, so the second condition is checked using the expression:

```
bubblestatus = 1
```

We can check if the left mouse button is pressed using the MOUSECLICK() statement which returns 1 if the left mouse button is down at the time the statement is executed. So our final condition is checked using the expression:

```
MOUSECLICK() = 1
```

The complete IF statement is coded as:

```
IF SPRITE COLLISION(1,2) = 1 AND bubblestatus = 1 AND
    ↳MOUSECLICK() = 1
```

Play pop sound

Playing our previously loaded sound file requires the line

```
PLAY SOUND 1
```

which specifies the ID of the sound file to be played.

Change “bubble” sprite to show popped bubble

Since the “bubble” sprite contains two frames, all that is required to show the popped bubble is to change the frame being displayed from the first (unpopped) to the second (popped) and this is done with the statement

```
SET SPRITE FRAME 2,2
```

which sets sprite 2 to show frame 2.

Set the status of the “bubble” sprite to “popped”

All that is required here is that we change the value held in the variable *bubblestatus* to 2:

```
bubblestatus = 2
```

UNTIL "bubble" sprite's status is "popped"

This is coded as:

```
UNTIL bubblestatus = 2
```

At last we are ready to produce the complete program (see FIG-1.3). Comments have been added to the code to aid readability.

FIG-1.3

Single Bubble Pop Program

```
REM **********************************
REM ***         Set up game        ***
REM **********************************

REM *** Set screen resolution      ***
SET DISPLAY MODE 1920,1200,32

REM *** Load sound file            ***
LOAD SOUND "pop.wav",1

REM *** Create mouse tracker sprite ***
LOAD IMAGE "black.bmp",1
SPRITE 1,640,480,1

REM *** Create and show "bubble" sprite     ***
CREATE ANIMATED SPRITE 2, "bubble.bmp",2,1,2
SPRITE 2,640,480,2
REM *** Set sprite state ***
spritestate = 1

REM **********************************
REM ***       Run game             ***
REM **********************************

REPEAT
   REM *** Move tracker sprite     ***
   SPRITE 1, MOUSEX(),MOUSEY(),1
   REM *** IF popping bubble       ***
   IF SPRITE COLLISION(1,2) AND bubblestatus = 1 AND MOUSECLICK() = 1
      REM *** Play sound           ***
      PLAY SOUND 1
      REM *** Show popped bubble   ***
      SET SPRITE FRAME 2,2
      REM *** Change bubble status ***
      spritestate = 2
   ENDIF
UNTIL spritestate = 2



REM *** End program                ***
WAIT 1000
END
```

Notice that two more lines have been added to the end of the listing which halt the program for one second before terminating.

Activity 1.3

Create the program shown in FIG-1.3.

Change the screen resolution to match your own set up.

Copy the three files used into the same folder as your program.

Run the program and check that it operates correctly.

The Second Version

Not satisfied popping just one bubble? In this next version we will create a sheet of 100 bubbles.

Of course, this makes the program a bit more complicated. New problems we have to think about include the need for separate status variables for every bubble, many more sprites, detecting which of the many sprites the mouse is over and detecting when all sprites have been popped.

To create and position the 100 bubble sprites needed we use the code:

```
FOR c = 2 TO 101
   CREATE ANIMATED SPRITE c, "bubble.bmp",2,1,2
   SPRITE c,400+((c-2) mod 10)*64,400+((c-2)/10)*64,2
NEXT c
```

The variable *c* determines the ID of each sprite, so the sprites are numbered 2 to 101. Since the sprites are exactly 64 x 64 pixels, each sprite needs to be exactly 64 pixels further to the right than the previous pixel and after 10 sprites have been positioned we need to go down 64 pixels to create the next line of bubbles. It is this requirement that gives the rather complex expressions for the sprite's *x* and *y* coordinates in the SPRITE statement. If you wish the sprites to start further to the left on your screen, reduce the first 400 in the SPRITE command; if you want the sprites to start further up the screen, reduce the second 400.

To create a status value for each sprite we need to set up an array with an element for each of the bubble sprites. Initially, every bubble is unpopped, so all elements should have their status set to 1. This requires the code:

```
DIM spritestate(102)
FOR c = 2 TO 101
   spritestate(c) = 1
NEXT c
```

Notice that only cells 2 to 101 are used. This corresponds to the IDs given to the bubble sprites which will make changing the status as a bubble is popped a little easier.

To check which particular bubble the mouse pointer is over we make use of a special version of the SPRITE COLLISION statement. Normally, we specify the IDs of the two sprites we wish to check, but if we specify the ID of only the first sprite (using zero in place of the second ID value) the SPRITE COLLISION command will return the ID of the sprite which has been involved in a collision with the first sprite. This allows us to

use the line

```
bubbleover = SPRITE COLLISION(1,0)
```

to determine which sprite the "black" sprite (sprite 1) at the tip of the mouse pointer has collided with and to store the result in a variable named *bubbleover*. If no collision has occurred, then *bubbleover* will be set to zero.

We can then use the value held in *bubbleover* to determine if a bubble has been popped and to change the image shown by the appropriate sprite using the code:

```
IF bubbleover > 1 AND MOUSECLICK() = 1 AND
    ↳spritestate(bubbleover) = 1
      PLAY SOUND 1
      SET SPRITE FRAME bubbleover,2
      spritestate(bubbleover) = 2
ENDIF
```

To determine when the 100 bubbles have been popped, we need to create a count variable (initialising it to zero) and incrementing it each time a new bubble is popped. The UNTIL statement would then check for this count reaching 100.

The complete code for the second version of the program is shown in FIG-1.4.

FIG-1.4

Sheet of Bubbles Program

```
REM **********************************
REM ***         Set up game        ***
REM **********************************

REM *** Set screen resolution      ***
SET DISPLAY MODE 1920, 1200,32

REM *** Load sound file           ***
LOAD SOUND "pop.wav",1

REM *** Create mouse tracker sprite ***
LOAD IMAGE "black.bmp",1
SPRITE 1,640,480,1

REM *** Create and show sprite     ***
FOR c = 2 TO 101
   CREATE ANIMATED SPRITE c, "bubble.bmp",2,1,2
   SPRITE c,400+((c-2) mod 10)*64,400+((c-2)/10)*64,2
NEXT c

REM *** Set sprite state ***
DIM spritestate(102)
FOR c = 2 TO 101
   spritestate(c) = 1
NEXT c

REM *** Set count to zero ***
count = 0

REM **********************************
REM ***       Run game             ***
REM **********************************
```

FIG-1.4
(continued)

Sheet of Bubbles Program

```
REPEAT
   REM *** Move tracker sprite        ***
   SPRITE 1, MOUSEX(),MOUSEY(),1
   REM *** Check for sprite collision ***
   bubbleover = SPRITE COLLISION(1,0)
   REM *** IF popping bubble          ***
   IF bubbleover > 1 AND MOUSECLICK() = 1 AND
      ↳spritestate(bubbleover) = 1
      REM *** Play sound              ***
      PLAY SOUND 1
      REM *** Show popped bubble      ***
      SET SPRITE FRAME bubbleover,2
      REM *** Change bubble status    ***
      spritestate(bubbleover) = 2
      REM *** Increment count         ***
      count = count + 1
   ENDIF
UNTIL count = 100    `stop when all bubbles popped

REM *** End program                   ***
WAIT 1000
END
```

Activity 1.4.

Type in and test the program given in FIG-1.4.