Creating a Splash Screen

A Tutorial for BlitzMax

Sloan Kelly

Introduction

This text will guide you through the creation of a splash screen that could be used in your programs. It shows a number of techniques that the reader might find helpful:

- Masked text area
- Moving highlight
- State machine
- Fading to black

By the end of this tutorial you will be able to create a splash screen for your game and understand some other concepts too.

The tutorial uses a mask image to simulate a light passing over chrome text with the highlight only affecting the text itself and leaving the black background untouched.

Getting Started

You will need BlitzMax installed before starting this tutorial. It is assumed that you are using the BlitzMax IDE.

Resources

The complete source code and images, as well as their Photoshop sources (PSD files) are located on my web site at http://sloankelly.net/. Click on the "Resources" link at the top of the home page.

If you are using your own resources, you should provide the following images.

Masked Text Image

The masked text image contains an outline of a piece of text:

Figure 1 - Example of masked text. The checkerboard represents the alpha of the image

The checkerboard is the *alpha* of the image. This is the part of the image that *will not be drawn* in BlitzMax. The file is called *mask.png* and is 600x72 pixels.

Background Image

The background text image can be anything, but I chose this chrome gradient because it suited the splash screen I was creating:

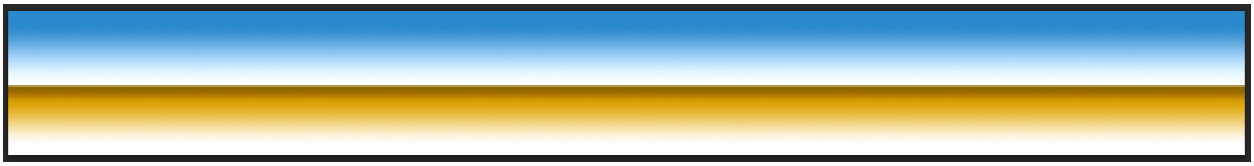

Figure 2 - The chrome-style background image

This will be displayed *underneath* the mask and will be displayed on the screen like so:

Figure 3 - The combined background and mask

The effect is to have the highlight pass over the text but not shine on the black background. To have the illusion that the light is being absorbed by the black and reflected by the chrome. The name of the background image file is *chrome.png* and is 600x72 pixels.

The Highlight Image

The highlight image is a white gradient on an angle, shown with a black background here for clarity. It too is on an alpha background:

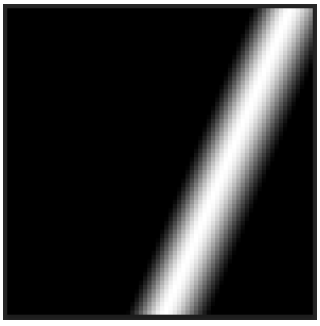

Figure 4 - The highlight

The highlight image file is called *highlight.png* and is 72x72 pixels.

Start BlitzMax

Start BlitzMax and create a new document in the same location as the .png files. Save it as *splash.bmx*. As we go, I will explain what each part of the code does. The code that you type into BlitzMax is shown in monospaced type inside a grey box.

The Code

Let's start with the basics first. Enter the following text into BlitzMax:

```
SuperStrict
Graphics 800, 600
SetBlend ALPHABLEND
```

SuperStrict puts the compiler into a mode that forces you to define variables before you use it. I like this mode because it means that I can end up with fewer bugs due to spelling mistakes.

I've chosen 800x600 for the graphics mode because that's a fairly common mode that graphics cards are capable of producing and because it's suitable enough for our purposes. The blend mode is set because we are using PNGs with alpha and I want to preserve the soft edges of the graphics.

```
Const STATE_CHROME:Int = 1
Const STATE_WAIT:Int = 2
Const STATE_FADE:Int = 3
```

These are the states that the splash screen uses;

- Show the chrome highlighter
- Wait a certain amount of time
- Fade to black

```
Const speed:Float = 300.0
Const fadeSpeed:Float = 85.0
```

I'm a huge proponent of using the pixels-per-second method to move sprites. The alternative is to assume that you will be running at 60 frames per second (or 30, or whatever) and then add a number each frame. Adding '4 pixels each tick' has a couple of drawbacks.

By using pixels-per-second method, rather than just adding 4 pixels each 'tick', you ensure that no matter how fast or slow your machine is, it will always travel at a constant speed. The best part is that even if the program skips a frame, this method will ensure that your characters will always be in the right place.

```
Local mask:TImage = LoadImage("mask.png")
Local chrome:TImage = LoadImage("chrome.png")
Local highlight:TImage = LoadImage("highlight.png")
```

The images are loaded into memory, you could embed them using the `IncBin` command, but in this example we'll just leave them on disk.

The images must be centred on screen so to ensure that, I set up some variables to hold the image positions:

```
Local x:Int = (800 - ImageWidth(mask)) / 2
Local y:Int = (600 - ImageHeight(mask)) / 2
```

The first group of co-ordinates holds the x- and y- positions of the mask and chrome images. I use the `ImageWidth()` and `ImageHeight()` to determine the width and height of the mask image. Since the mask and the chrome images are the same size this will display both images centre screen.

```
Local hx:Float = x
Local hy:Float = y
```

The next group of co-ordinates is the position of the highlight as it moves across the text. These will be updated each frame. Talking of timing, the next two variables are to control the timing of various events in our splash screen:

```
Local lastTime:Int = MilliSecs()
Local waitTill:Int
```

We need to control our colour intensity to make it look like our image is fading, this intensity is controlled by these variables:

```
Local r:Float = 255, g:Float = 255, b:Float = 255
```

Each represents the red, green and blue components of the draw colour and we will use `SetColor()` to update the drawing colour.

Our final variable holds the current *state* of the splash screen. The state is initially set to STATE_CHROME:

```
Local state:Int = STATE_CHROME
```

For more information on state machines, see the section on Finite State Machines below.

That's it for our constants and variables section of the program, now we move onto the main loop:

```
While Not KeyHit(KEY_ESCAPE) And r <> 0
```

The while loop ensures that the program still runs until the user presses the escape key, or the colours have faded all the way down to black.

```
	Cls
```

Before you draw anything to the screen you should clear it first, unless you are wanting to do some special effects with an un-cleared screen. The Cls command clears everything that is on the current back buffer[1] ready for us to draw on it.

```
	Local timeDiff:Int = MilliSecs() - lastTime
	lastTime = MilliSecs()
```

This is a common technique to get the *delta time* (the time between now and the last time this piece of code was run) of the tick. We can use this delta time to update animations, move characters, fade colours etc. The `MilliSecs()` function returns the number of milliseconds since the game started.

[1] http://en.wikipedia.org/wiki/Framebuffer#Page_flipping

Each time we go through the while loop, we check to see what state we're in. For this we use `Select`.

```
      Select state
            Case STATE_CHROME
```

Our first state is the 'chrome' state that will move the highlight accent across the face of the text.

```
                  Local distance:Float = (Float(timeDiff) / 1000) * speed
                  hx:+distance
```

We use the formula to determine distance; `distance = speed * time`. In this case, the time is the delta value and the speed is the number of pixels per second 'speed' that we created as a constant above.

```
                  If hx > (x + (600 - 80))
                        state = STATE_WAIT
                        waitTill = MilliSecs() + 3000
                  End If
```

At the end of this state we perform a *state transition* if the highlight has reached the end of the text. We set the state variable to the next state and set the `waitTill` to be three seconds (3000 milliseconds) in the future.

```
            Case STATE_WAIT
                  If MilliSecs() > waitTill
                        state = STATE_FADE
                  End If
```

The wait state holds the text on-screen for three seconds (`waitTill` was set in `STATE_CHROME`). When the three seconds are up, the state is changed to `STATE_FADE`.

```
            Case STATE_FADE
```

The last state that the splash screen goes through is the fade to black stage. In order to simulate this, we're going to use calls to `SetColor()` to set the draw colour of the *subsequent draw operations*.

```
                  Local change:Float = fadeSpeed * (Float(timeDiff) / 1000)
                  r:-change
                  g:-change
                  b:-change
```

The change of intensity is calculated in much the same we as we did the distance travelled by the highlighter. The fade speed is multiplied by the delta time. The delta time in `timeDiff` is in the range 0..1000, so we must make that a fraction by dividing that number by 1000.

Next we make a check to see if the red component is less than zero. Because all the components have the same value, we don't have to check green and blue as well. If you want to do any special effect, like a blue fade like they used to do in older games, you'd have to re-write this fader. Because it's just a simple fade to black, we can leave the if-check as this:

```
                If r < 0
                      r = 0
                      g = 0
                      b = 0
                End If
      End Select
```

And that ends our state machine. For the most part, this is how a lot of state machines are created; a control variable and a `Select` statement.

The main drawing code displays the images in a distinct order. Images drawn first are drawn at the back. Subsequent images are drawn on top of the previous images.

```
      SetColor r, g, b
```

This sets the intensity for the subsequent calls. This is our fade colour, over time in state STATE_FADE this becomes black.

```
      DrawImage chrome, x, y
```

Draw the chrome image at the specified intensity.

```
      SetColor 255, 255, 255
```

We don't want any other draw function to use the fade intensity, so we reset the draw colour to full-brightness.

```
      DrawImage highlight, hx, hy

      DrawImage mask, x, y
```

The highlight is drawn first then the mask. This means that highlight is drawn *underneath* the mask image that gives the illusion that the highlight is just affecting the text. Try flipping the order of highlight and mask and you'll see what I mean.

```
      Flip
```

The `Flip` function swaps the back buffer to the front buffer. Everything we've just drawn is now displayed to the user.

```
EndWhile
```

Don't forget your `EndWhile`!

Conclusion

Make sure that you have typed in everything as-is and run the program. You should see the chrome text and a highlight going across it. After a short time (3 seconds), the text will begin to fade to black. At the end of the fade, the program will terminate.

Don't forget to download the graphics from the web site, http://sloankelly.net/.

Additional Exercises

There is a fade down state, but no fade up state. Implement a fade up state. Hint: You will need to create a new state called `STATE_FADEUP` and write a new `Case` block in the `Select` statement.

Appendix – Finite State Machines

A state machine is a program, or robot or subroutine or whatever, that can be placed into any number of 'states of being'. They are called 'finite' state machines because there are a fixed number of states. For example, an elevator has two states; moving to a floor and stopped. To move between these states you apply some rule during the current state and then *transition* to the next state if that rule is met. For example:

An elevator remains in the STOPPED state until a guest presses a floor button. At that time the elevator changes to the MOVING state until the floor is reached. At that time, the elevator once again enters the STOPPED state.

We can make this example more complex with call buttons and multiple floors, etc. But let's just use this simple elevator just now. We can then draw a state transition diagram to describe the states and the state transitions:

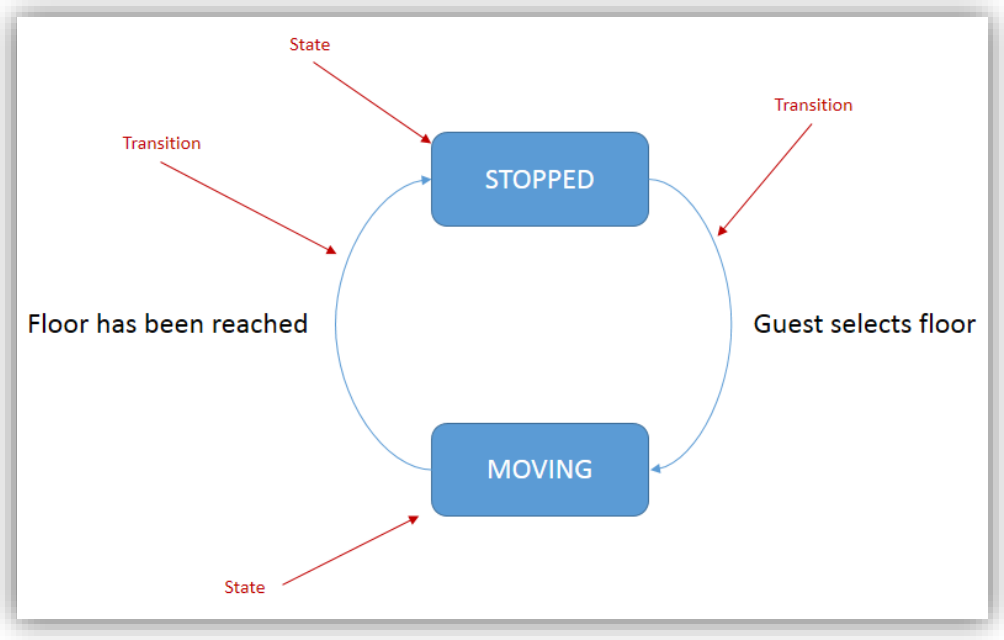

Figure 5 - State transition of an elevator

The states are represented by the blue rectangles, the state transitions and their rules are denoted by the blue arrows. Applying this to our splash screen, our state diagram looks like this:

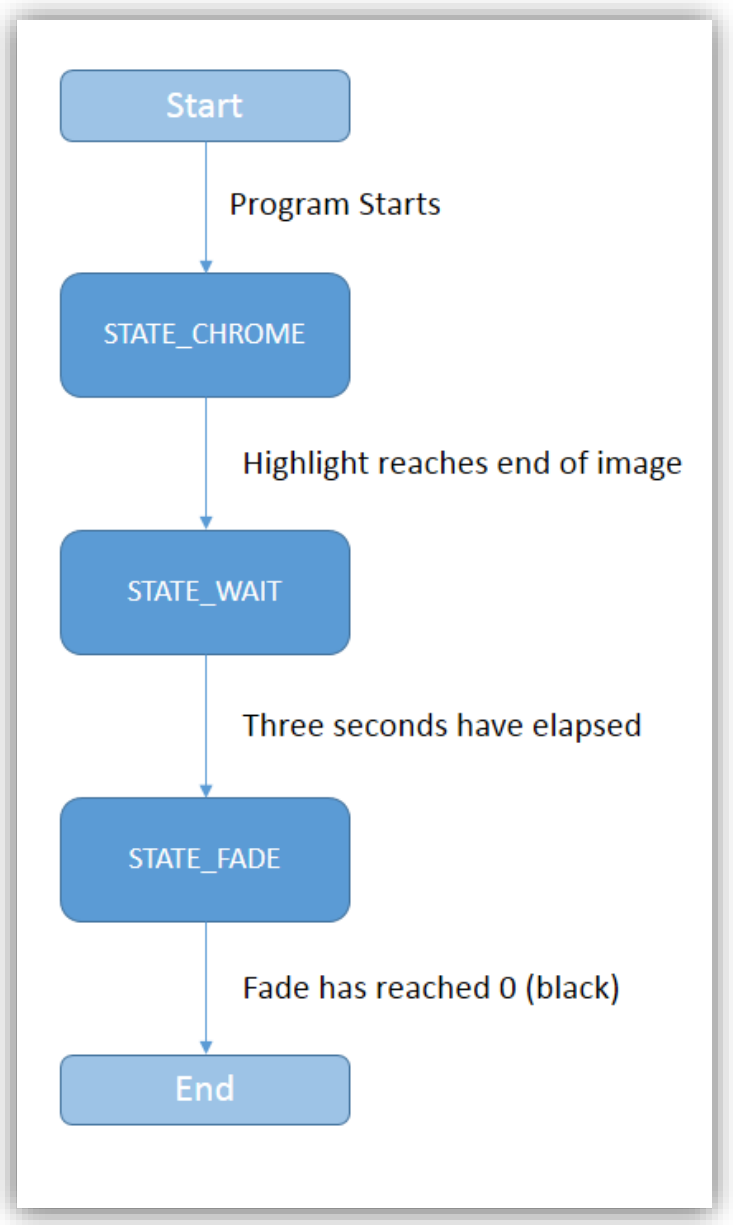

Figure 6 - State transition diagram for the splash screen

Our state machine is even simpler because it just performs a linear progression through the states.

Finite state machines can be used for a variety of purposes:

- Animation
- Artificial intelligence
- Program flow
- Menus

Animation

A character can be in a variety of states; running, walking or swimming. Each has its own actions and possible inputs. For example, in Super Mario when you press the jump button when walking he does a small jump, when running he does a longer jump and when swimming he swims closer to the surface. Each action from that one button is determined from what *state* the character is in.

Artificial Intelligence

Until recently, most AI in computer games used finite state machines. For AI, state machines are used to keep an enemy guard patrolling a specific area. If they 'hear' or 'see' the player, they will engage. The initial state might be set to PATROL, but on seeing a player they might transition to the ENGAGE_WITH_RANGED_WEAPON state.

Program Flow

Games themselves have different states and each part is separate;

- Splash screen
- Menus
- Playing the game
- Dying/game over screen
- …

Menus

Menus can have a sub-set of a state machine, for example;

- Video and audio settings
- Controls
- Language options

So you can see that state machines aren't just for artificial intelligence, they're everywhere in a game.