Creating A Slide Puzzle

In a Windows Environment with C# and WPF

Louis D'hauwe

27 November 2014

odisee

Introduction 3
Bitmaps 4
Displaying a Bitmap 4
Manipulating pixels 4
OO-fying pixels 5
Game setup 6
Animations 7
Functions 8
Gaussian Blur 9
Optimisations 11
Shadows 12
Conclusion 12

Introduction

In this paper I will discuss the making of a slide puzzle game in a Windows environment. In case you're not familiar with this kind of puzzle: it's a puzzle where an image is broken in tiles and shuffled randomly. The goal being to restore the original image.

I obviously won't be discussing the physical making of such a puzzle, but rather a digital version. To make this in a Windows environment with C# we basically have 2 choices. We can either use Windows Forms or Windows Presentation Foundation (WPF). Windows Forms being the older of the two, with less support for eye candy, doesn't seem like a viable option for this kind of application.[1]

One more thing to note is that this is an exercise focused on bitmap manipulation. Meaning we will avoid using many built in functions in the .NET Framework and rather explore the code that goes behind such operations. This also means we'll face quiet the overhead for doing pixel per pixel manipulation. If this game were to be made for a client, we would probably just cut the image in pieces (our tiles) and move those image views.

[1] http://en.wikipedia.org/wiki/Windows_Forms#History_.26_future

Bitmaps

Displaying a Bitmap

Before we can make our game, we need a way of showing an image on screen. In C# terms, this means displaying a Bitmap object. In WPF we can use the Image control. This control can display a BitmapImage object.[2] But since we would like to keep backwards compatibility with Windows Forms, we want to use the Bitmap class.[3] This will require us to convert a Bitmap to a BitmapImage object for displaying in the WPF application. This turns out to be trickier than expected, but the System.Windows.Interop.Imaging class can help us here.[4]

Manipulating pixels

Now that we can display an image, we need a way of manipulating the pixels of a Bitmap object. We could use the built in methods SetPixel(int, int) and GetPixel() for this, but those can't give us the performance we want for this type of application.[5] The way the SetPixel method works is by what's called "locking" the pixels' bytes from the Bitmap object to access and change one pixel. Locking means we "lock" the bits in memory (of a Bitmap, in this case) so we can alter the bits programmatically.[6]

After a change has occurred those pixels will be unlocked again (for safety purposes). We can imagine an image of 300 by 300 pixels where we have to slide a tile in our game. If we have a 3 by 3 grid for our game this means updating 20.000 pixels (100 x 100 x 2). Unlocking and locking the bitmap 20.000 times is unnecessary, since we only need to display the image once all 20.000 pixels are updated.

A much faster way is to manually lock the pixels once, change the 20.000 pixel values and display our image. This way, we don't even need to unlock the pixels again after each game move. Since we can simply unlock the bits once the game is over. In order to easily lock and unlock we can use the methods of the Bitmap class LockBits and UnLockBits.[7] [8]

---

[2] http://msdn.microsoft.com/en-us/library/system.windows.media.imaging.bitmapimage(v=vs.110).aspx

[3] http://msdn.microsoft.com/en-us/library/system.drawing.bitmap(v=vs.110).aspx

[4] http://stackoverflow.com/questions/1118496/using-image-control-in-wpf-to-display-system-drawing-bitmap

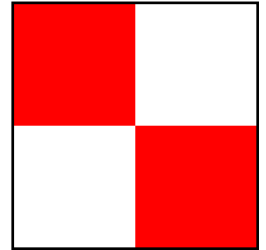
[5] http://msdn.microsoft.com/en-us/library/system.drawing.bitmap.setpixel(v=vs.110).aspx

[6] http://msdn.microsoft.com/en-us/library/5ey6h79d(v=vs.110).aspx

[7] http://msdn.microsoft.com/en-us/library/5ey6h79d(v=vs.110).aspx

[8] http://msdn.microsoft.com/en-us/library/system.drawing.bitmap.unlockbits(v=vs.110).aspx

OO-fying pixels



Once we lock the bits of a Bitmap object we can access its pixel values. With the System.Runtime.InteropServices.Marshal.Copy method we can copy those values to an array. There's a bit of a disappointment here, however. That method will only copy to a one dimensional array. This array will contain bytes (meaning a range from 0-255) that each represent one color channel of one pixel. Imaging having a bitmap of 2 x 2 in the ARGB color space. Each pixel would take up 4 bytes (4 color channels), so 32 bits for each pixel. For the example shown on the right, that byte array would look like this:

```
[0, 0, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 0, 0, 255, 555]
```

Already, this seems very unreadable. But let's dissect it anyways. The color space ARGB is represented in reverse order (meaning BGRA). So the first pixel has a red and alpha channel value of 255 and blue and green of 0. A better representation would be in a two dimensional array, giving us the following:

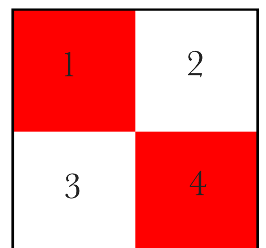
```
[[0, 0, 255, 255],

[255, 255, 255, 255],

[255, 255, 255, 255],

[0, 0, 255, 555]]
```

This is nicer, but not perfect yet. What we would really want is that we have an array containing arrays where those lather arrays each represent one line in our bitmap. To do this with the current "raw" bytes, we would need to use a three dimensional array. To avoid this, and to objectify this, we can simply make a class "ARGBPixel" for this. This class will hold 4 bytes, with nice accessors and constructors. With this, we can become the following array:




```
[[pixel1, pixel2],

 [pixel3, pixel4]]
```

This approach works ... for ARGB pixels. But once we use an RGB color space for a bitmap we find something unexpected. For performance reasons, Microsoft has implemented the Bitmap class so that the one dimensional array we talked about earlier rounds each line up to 32 bits (= 4 bytes). This basically means that for the 4 pixel image from earlier in RGB we get this:

[0, 0, 255, 255, 255, 255, ***0, 0***, 255, 255, 255, 0, 0, 255, ***0, 0***]

First thing to notice is that the order of bytes here is BGR. More importantly are the 4 zeros marked. Those are junk bytes, meaning they don't represent any pixel information from the bitmap, but are merely there to round the line upwards to 32 bits.

We can calculate the number of junk bytes on each line as following:

```
junk bits = (32 - ((Width * bitSize) % 32))

junk bytes = junk bits / 8
```

For our example with RGB we can calculate:

```
junk bits = (32 - ((2 * 24) % 32) = 32 - (48 % 32) = 32 - 16 = 16

junk bytes = 16 / 8 = 2
```

With this formula, we can also conclude that an image in the ARGB color space will never have junk bytes. For sake of being complete, the official name of the number of bytes on each, including the junk ones, is called the stride.[9]

Once we know this, we can implement this to filter out the junk bytes. However, if we want to show the image on screen we have to encode our image from our two dimensional array with OO structure to a one dimensional array. This will require us to add the junk bytes (if any) back at the end of each line.

Game setup

Now that we have a good understanding and structure for manipulating bitmaps we can easily build our slide puzzle game. But first things first, imagine we have puzzle with 9 tiles (3 by 3). We would like to have an easy way of knowing where each tile is at a given moment. Furthermore, we need a way of checking if the puzzle is solved. To do this (again, in an OO way) we will need a model class, that basically wraps a two dimensional integer array. For our 3 by 3 tiles example the array would look like this:

```
[ [0, 1, 2 ],
  [3, 4, 5],
  [6, 7, 8] ]
```

Why a wrapper class? So we can easily add help methods, such as a SwapIndices method. And an Equals method solves our problem for checking if the puzzle is solved. In a slide puzzle game there's always one tile that's empty (so you can slide the other slides). To easily recognise this tile, we would use a constant, such as -1. In the example above, index 2 would become -1. This way we know that any index next to -1 is a tile that can slide. So when we swap -1 and 5 we know that tile 4, 5 and 8 can slide.

[9] http://msdn.microsoft.com/en-us/library/system.drawing.imaging.bitmapdata.stride(v=vs.110).aspx

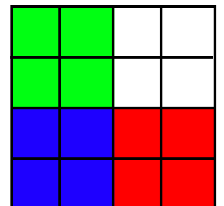
```
[ [0, 1, 5 ],
  [3, 4, -1],
  [6, 7, 8] ]
```

At most, 4 tiles can slide at a given moment. And at least 2 (corner case). For moving the tiles in our bitmap we would simply calculate width and height of a tile, and with some simple for loops, we can swap a tile.

There is one catch, though. Before you start a game, you need to check if the width and height of the bitmap you're using is divisible by the number of columns and rows respectively. If you were to have an image of 320 by 320 pixels and want to play a slide puzzle game of 3 by 3 tiles. You'll notice that 320 isn't divisible by 3. This will give you 2 pixels unused at each line, since 320 % 3 = 2. The solution to this problem is easy: cropping the bitmap to a width and height that's divisible by the number of columns and rows respectively.

Animations

We could wrap up here, and we would have a puzzle game that works. But to make it more interesting, let's animate the movement of the tiles. This seems like a hard task to do pixel per pixel. So let's go over it step per step. First, we need to understand what exactly happens when we swap 2 tiles.



Let's take a bigger example than before, this time we'll have 4 by 4 pixels and 2 by 2 tiles. If the white tile represents the empty one, that means the green and red one can slide. So how would the red tile exactly slide where the white one is?

I've numbered the pixels to make this easy.

		1	2
		3	4
		5	6
		7	8

Step 0

		5	2
		3	4
		1	6
		7	8

Step 1

		5	6
		3	4
		1	2
		7	8

Step 2

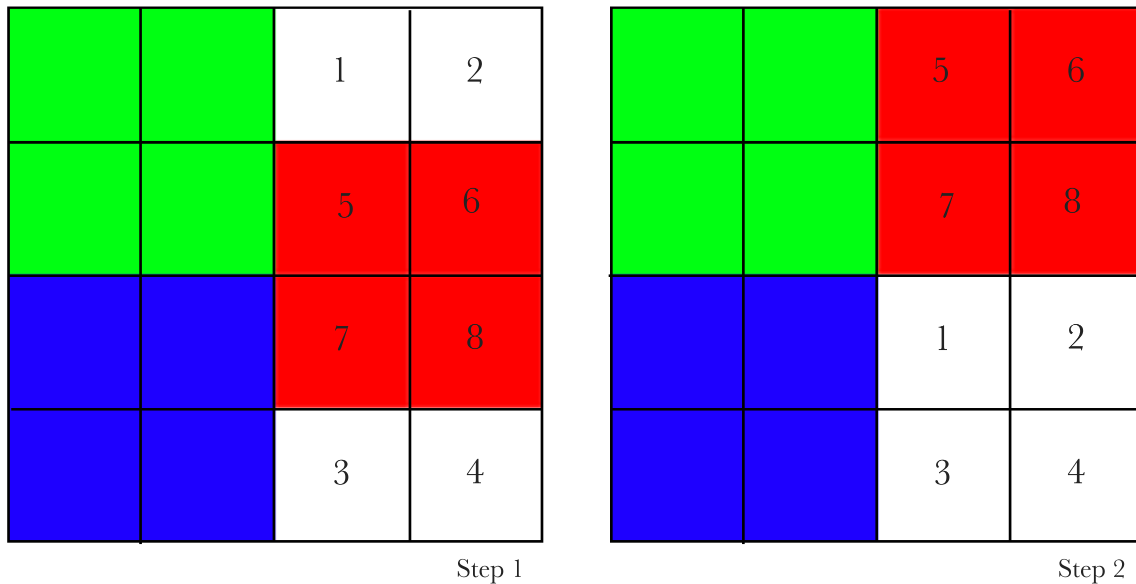
		5	6
		7	4
		1	2
		3	8

Step 3

		5	6
		7	8
		1	2
		3	4

Step 4

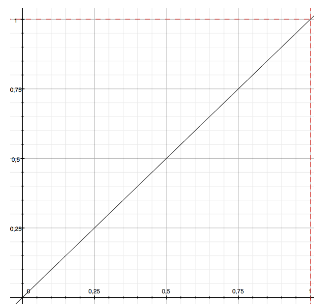
In practice we will of course go from step 0 to step 4, not seeing the steps between. For a tile slide animation we wouldn't want those steps shown, though. What we **do** want, is this:



In order to do this, we need a frame stack. In C# we can use the Queue class. This has a FIFO principle (first frame in, first frame out).[10] Once we have this, we just hook up a timer at 60 FPS (meaning an interval of 1000/60 ms = 16ms). At each interval we pop a frame. When the user makes a move, you add all frames for that move to the stack. This has an interesting disadvantage: a new move move can only be animated once the previous one is finished. In other words, the frame queue has to be empty. This can cause for a delay between moves, so it's best to keep animation times short.

Functions

In the world of animations, there's more than just a linear (which is what we have now) animation. Linear just means the animation moves at a constant interval. A nicer animation function can be become when we use easing. But first, let's define how animation functions work.

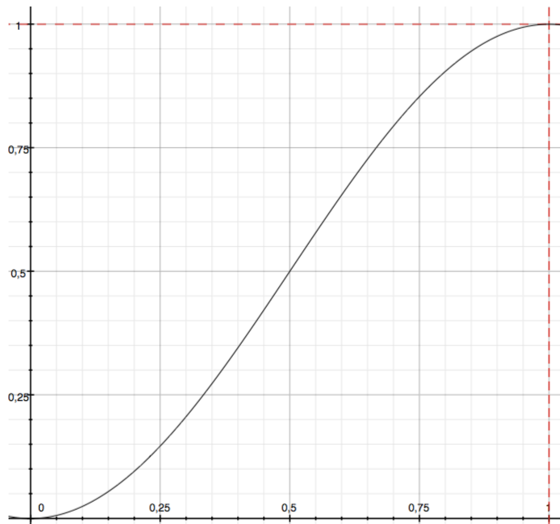


A linear function is just the function y = x. More important to understand is the fact that the boundaries for these animations are [0, 1] on the x and y axis. The x axis is the progress of the animation (in time). The y axis is the animation percentage. For instance: an animation that moves an object from (0, 0) to (100) would be at (0, 50) at 50% of the animation (y = 0.5).

[10] http://msdn.microsoft.com/en-us/library/system.collections.queue(v=vs.110).aspx

Since y = x, (0, 50) will occur at 0.5 of x as well. An ease in out function has the following formula: $y = \frac{\cos((x+1)\pi)}{2} + 0.5$ [11]

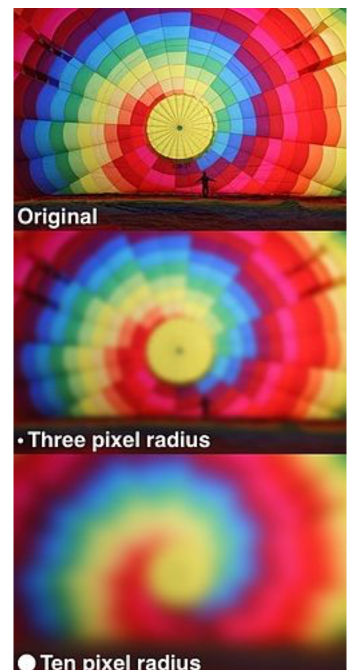
Giving us the following graph:



We can see that in this function the animation still reaches (1, 1) in the same amount of time as with the linear one. But it decelerates at the start, and accelerates at the end. Giving it a more natural feeling.

Gaussian Blur

A gaussian blur is an advanced type of blur that will take the neighbours of the neighbours of pixels in account to calculate the blur. [12] It can be used to remove image noise, but we'll use it as a way of showing our users that their time left for solving the puzzle is nearing zero. The great thing about a gaussian blur is that it takes a parameter for the diameter (or radius) for the blur. This means we can animate a blur with a diameter from 0 to 51. Notice that the diameter has to be an odd number. This is important to calculate the kernel.



For a gaussian blur on our game image we will need the following formula:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

x and y = distance from origin (horizontally and vertically)

One variable we don't have yet for this formula is sigma (σ). For that we use the following formula:

σ = 0.3 * (radius - 1) + 0.8 [13]

[11] http://gizma.com/easing/

[12] http://en.wikipedia.org/wiki/Gaussian_function#Proof

[13] http://docs.opencv.org/modules/imgproc/doc/filtering.html#getgaussiankernel

Let's say we want to blur an image with a 5 pixel diameter (2 pixel radius). We then first need the kernel using the above formula, giving us this:

0.00482336939763721	0.016662191163091	0.0251879395577935	0.016662191163091	0.00482336939763721
0.016662191163091	0.0575590612013644	0.0870110143589512	0.0575590612013644	0.016662191163091
0.0251879395577935	0.0870110143589512	0.131533010819748	0.0870110143589512	0.0251879395577935
0.016662191163091	0.0575590612013644	0.0870110143589512	0.0575590612013644	0.016662191163091
0.00482336939763721	0.016662191163091	0.0251879395577935	0.016662191163091	0.00482336939763721

We can see how the values are mirrored horizontally and vertically in the kernel, showing the true magic of a normal distribution.

A small, but important, detail is that if we add all these values we don't get 1.0. We get "0.96315607818746118". As a matter of fact: as the radius of our kernel gets larger, its sum will near 1.0, but only reach it at infinity (theoretically).

This brings the following effect: imagine having an ARGB image we blur with an alpha channel of 255 everywhere. That alpha channel will become 244 (255 * sum) after the blur, making the image slightly transparent. The other channels will suffer from this, too. Those will near towards 0, making the image darker.

To prevent all of this, we simply divide each value in the kernel by the total sum, making sure the new sum equals 1.0. Doing so, we get this:

0.00500787931143432	0.017299575365238	0.0261514619782019	0.017299575365238	0.00500787931143432
0.017299575365238	0.0597608866360303	0.0903394749090874	0.0597608866360303	0.017299575365238
0.0261514619782019	0.0903394749090874	0.136564585739081	0.0903394749090874	0.0261514619782019
0.017299575365238	0.0597608866360303	0.0903394749090874	0.0597608866360303	0.017299575365238
0.00500787931143432	0.017299575365238	0.0261514619782019	0.017299575365238	0.00500787931143432

Once we have a kernel, we can make a blur. Simply looping through each pixel and calculating the weighted average of the pixel and its neighbours, and this for all channels.

One disappointing conclusion we can make after doing all of this is that generating a gaussian blur is slow. Luckily, there are some performance optimisations we can do.

Optimisations

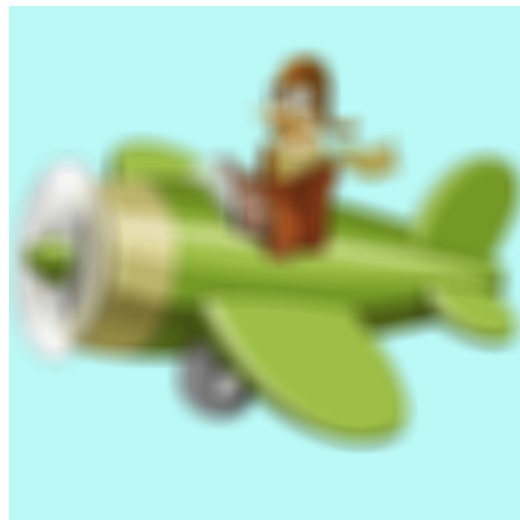
First of all, we can calculate the kernel once, and use it later. Instead of calculating it at each pixel. An even better optimisation is combining 2 motion blurs to become our blur.

A motion blur is the same as a gaussian blur with 1 difference: it's in one dimension.

Using the one dimensional formula, we can become a one dimensional kernel.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$ [14]

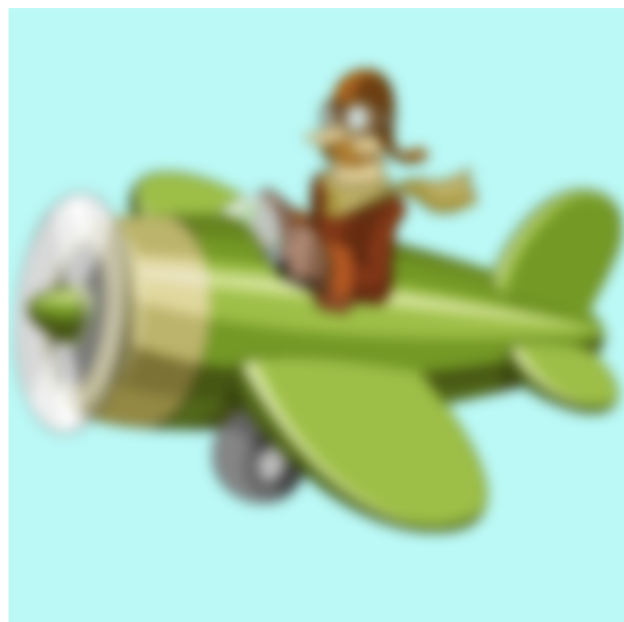
Vertical motion blur



Horizontal motion blur added

This is a much faster gaussian blur, but at a cost: it has a slightly worse quality than a 'real' gaussian blur.

One last improvement is an obvious one. Pre rendering the blur animation for our timer. And doing this on a background thread. This way we can render our blurs while the user is playing. As long as we pick a reasonably timer and reasonable blur parameters, the blurs should finish before the users gets to the



Full gaussian blur

[14] http://en.wikipedia.org/wiki/Gaussian_blur#Mechanics

part where the image starts blurring (for example: last 20 seconds of the timer).

A more definite performance improvement would we to use the GPU to do the heavy lifting, meaning gaussian blurs are a lot faster when running on OpenCL.

Shadows

An interesting side track for our blur knowledge is rendering drop shadows. As it turns out, drop shadows are just gaussian blurs of black pixels. A couple of thing to note: The shadow will probably exceed the bounds of the original bitmap it's generating a shadow for. So we have to make sure the shadow isn't clipped.

An optimisation that's unique to generating these shadows is the fact that, since there are only black pixels, we can work in greyscale. Meaning we have 2 less channels to calculate (ARGB => AG).

Conclusion

The approach taken for the creation of the game in this paper doesn't appear to be a realistic one for a consumer release. Once we get to images that are 1000 by 1000 pixels or larger, even a very fast i7 processor will get in trouble. The use of the GPU is essential in these type of applications.