Python Background Creator

A User's Manual

Kevin L. Biewesch

April 12, 2018

You like to create a series of images to use as a background for your presentation? This program makes it easy to create a couple of them in a row all the while having the same design you have come up with!

Contents

1. Who needs this?

The main purpose of this program is to create a row of different backgrounds for your presentations, be it with PowerPoint or LaTeX's Beamer, and to use your hand-crafted themes instead of the predefined ones that are available here and there.

If you are someone like me and always try to deliver something visually impressive or creative then this may be what you have been looking for! I like to use images as my background, as they provide *maximum* flexibility. You can, for example, easily create a simple theme which is only made of a blank body and have the title be written on a coloured box, kind of like the default ones. Or you can go all out and use actual pictures! You can find some examples of mine in the appendix section A on page 11.

2. How to use this program?

The aim is to make this program relatively *easy to use* compared to the fiddling you would have to go through if you wanted to create themes of the likes of my examples.

To kick things off you have to give the program an image to work with. Next you can give specifications that control the look and effects applied to the different parts of your background. And you are done!

This may sound pretty useless, and it probably is if you just try to create a very simple background without any special effects. Or even if you have special effects going on but use the same image all throughout your presentation. In that case I recommend using some image editing software like Gimp or Krita for example, to name two free alternatives. If on the other hand you would like to go out of the box and have different images for the different topics then it can become quite hard and annoying to maintain a certain regularity within your background images and that is where this program comes in handy. The commands are explained in section 5 when we are Getting fancy ...

3. Requirements

This program has been written in Python 3[1] so you will need that to start off. Apart from this you will need the Python Image Library since this program makes use of this library, more specifically the Pillow[2] fork. The following lines have been used to import the library (but you don't need to include them in your code):

```
from PIL import Image
from PIL import ImageFilter
```

4. The image layout

The following figure 1 illustrates how the image is set up and what the diffrent components are. This will help you understand what the different parts of the image are. The corresponding annotation is in table 1 on the next page.

[1] https://www.python.org/downloads/
[2] https://github.com/python-pillow/Pillow

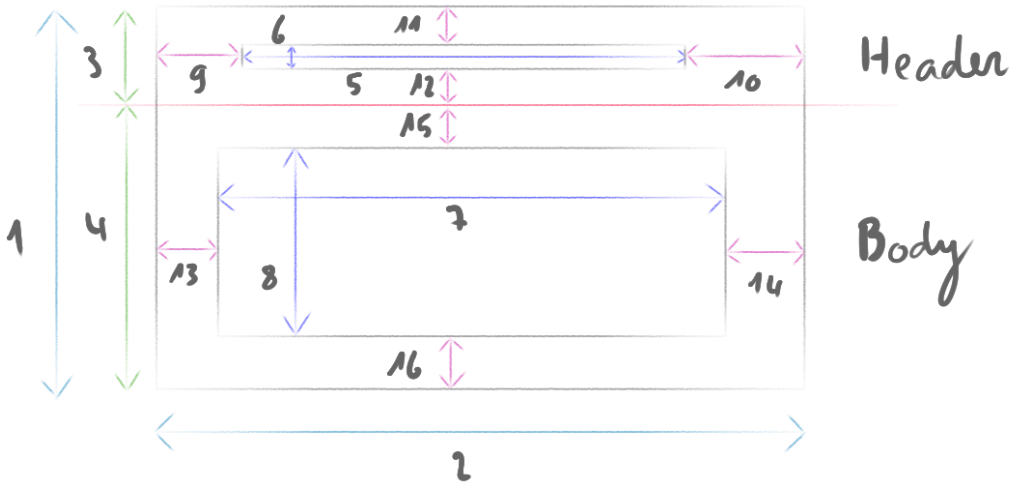

Figure 1: Drawing of the image layout

Item	Element
1	Image height
2	Image width
3	Header height
4	Body height
5	Header contents width
6	Header contents height
7	Body contents width
8	Body contents height
9	Header margin left
10	Header margin right
11	Header margin top
12	Header margin bottom
13	Body margin left
14	Body margin right
15	Body margin top
16	Body margin bottom

Table 1: Annotation of figure 1

Here illustrated are the different sizes you encounter in the image. You will likely never get in contact with them directly but it is always good to know what you are dealing with. In section 5 we will see how you can access these areas which should be pretty easy.

5. Getting fancy ...

Now we will finally have a look at actually writing our magical code.

5.1. Open an image

To start off, you should know that all the sizes in figure 1 on the preceding page are calculated once you create the image object. In order to work with an image with `pbc` you have to import the library first:

```
import pbc
```

and then open an image the following way (without and with path specification):

```
pic = pbc.Background('my_picture.jpg')
pic = pbc.Background('path/to/my_picture.png')
```

`pic` is here the image object on which you will be working. Of course you can name your object whatever you want, but I will stick to `pic` for now. The magic is happening after `Background()` returns the object. Now all the sizes are set. If you want to have different sizes then you will have to alter them here. We will get to this in section 5.3 on page 7.

For now, the only necessary parameter is the name of the image you want to work with. In the case where the image does not exists, either because you mistyped something or it really doesn't exist, or the image format is not supported then `PIL`[3] is the one to complain.

If the image is not located within your working directory then you can add the path to the image. Another option would be to change the working directory before opening the image using the `os` library:

```
import pbc
import os
os.chdir('path/to/')
pic = pbc.Background('my_picture.png')
```

Notice that `pbc` precedes `Background`. This is Python's way to access the Background class from the pbc module. You can avoid typing `pbc.Background` by importing the library this way:

```
from pbc import *
pic = Background('my_picture.jpg')
```

[3] `PIL` is the image library around which this program has been written.

5.2. The different methods

Here are listed and explained the different methods that exist to alter your image. They all have a similar way of use which makes them hopefully easy to use.

For each method its corresponding arguments are specified alongside their default values. If you use the following methods without any arguments then the default values will be taken.

Notice that almost every method has the arguments `part` and `region`. These arguments specify on which part of the image a certain action is performed. In order not to have to specify their values each time, here is a table with their possible values.

part	region		
full	full	/	/
header	inner	/	/
body	left	left-incl	left-excl
	right	right-incl	right-excl
	top	top-incl	top-excl
	bottom	bottom-incl	bottom-excl

Table 2: `part`, `region` values

You can mix and match all the part values with the region values in the table. Please note that if you chose `full` as your part then `inner`, `left`, `left-incl`, `left-excl`, `right`, `right-incl` and `right-excl` in the region column will produce no usable result. It just would not make much sense, therefore you will simply be affecting a small box in the top left corner of your image.

5.2.1. filter

The `filter` method lets you apply a couple of, guess what, filters.

```
pic.filter(operation='blur', part='full', region='full',  value=6)
```

The `operations` include:

None Does visually nothing

'raw' Paste the input image without any effects onto the desired area

'blur' Blur the desired area

'maxFilter' Apply a sort of painted look

'minFilter' Similar to the former but it looks slightly different

The `value` is only used to determine the intensity of the blur. Other than for the `blur` operation this parameter has no effect.

5.2.2. overlay

With this method you can *overlay* images, with or without transparency, to make things more interesting and simple evenly coloured images, with or without transparency, to lower contrast.

```
pic.overlay(ov_image='blank', part='full', region='full')
```

The overlay image `ov_image` argument can contain one of the following:

'blank' This will overlay an even grey colour

(tuple) With a *tuple* you can specify an even colour to be overlaid by its RGB or RGBA values. Either as (R,G,B) or as (R,G,B,A) for alpha values. The values range from 0 to 255.

'image' This will overlay an image. Insert a string of the *image* name and if necessary prepend the path to the image (just like you were to create an image object with `Background()` as shown in section 5.1 on page 4)

Please note that if the image you enter is larger than the area then the image will exceed the right and bottom boundaries of the area.

5.2.3. image

This method provides a few more options for overlaying images.

You can very well overlay images with the `overlay` method, but there the images are simply being put ontop of the canvas. If the image you give as input is larger than the desired area, then the image will exceed the boundarys.

Here you don't have to hand tailor your image to fit into the area, instead this method crops the image for you if you like. You can also set an anchor for the image's upper left corner. This way you can *move the image around* inside your area, but only if `borders` is set to `True`.

```
pic.image(picture=None, part='full', region='full',
          borders=True, anchor=None, transparency=255)
```

The following arguments' purpose are:

picture Specify the image just like in the `overlay` method. If set to `None` it does visually nothing.

borders Whether (`True`) or not (`False`) the image should be cropped in order to prevent it from exceeding the area's boundaries

anchor If `borders` is set to `True` you can place the upper left corner of the input image wherever you want. This way you can *move* the image around inside the area. Insert a tuple `(x,y)` specifying the location.

transparency The transparency value for the image. As of now, this option does not seem to work properly.

When setting the `anchor`, be aware that if the upper left corner is inside your area then the area will not be filled completely, but instead there might be space to the left or to the top of the area. The extrem case would be if you move the anchor farther than the right or bottom side of the area. In this case, the image will not even be visible.

5.2.4. margins

With this method you can toggle whether margins are included or excluded. By default they are included.

```
pic.margins(switch=None)
```

The `switch` argument can take the following values:

None Automatically toggles between including and excluding margins

True Include margins

False Exclude margins

When margins are included this means for example that whenever you use `left` as your region, it will actually use `left-incl` and when margins are excluded it will use `left-excl`. This happens under the hood and you can of course specify the long form for including or excluding margins, that is `left-incl` or `left-excl`. If for example you want to exclude all margins it can be quite tedious to write it out every time.

5.2.5. dimensions

Return the size of the specified area.[4]

```
pic.dimensions(part='full', region='full')
```

This method returns a tuple containing two tuples. The first tuple represents the size of the area and the second tuple the upper left corner of the area. The output is of this style: `((width, height), (x, y))`

5.3. Alter image dimensions

As previously mentionned in section 5.1 you can only alter the various image sizes when creating the object. There are two major reasons as to why you cannot alter the sizes on the fly. First reason being that the way the code has been written would make it a pain in the butt to change sizes on the go. I tried it and quit before I was done because the code had already doubled its size. The second reason is that it just wouldn't make much sense to change dimensions on the go because the margins might not fit together any more.

Here are all the arguments that are available to create a new image object:

```
pic = pbc.Background(image_name, **margins)
```

[4] It is not actually printing the result to the console.

This is kind of like what it looks like in the actual code. You can see that `image_name` is the image you enter and the `**margins` argument is a special one called keyword arguments. It allows you to enter an arbitrary amount of arguments alongside their values. This is how you can specify the margin sizes. There are two different ways to tackle this and both will be presented.

If the value entered is smaller than 1 then the margin thickness is expressed as a relative value. If the value is greater or equal to 1 then the thickness is expressed as the number of pixels. Below are listed the diffrent keywords which you can use alongside their default values

```
header_left = 0.1
header_right = 0.1
header_top = 0.2
header_bottom = 0.2

body_left = 0.15
body_right = 0.15
body_top = 0.1
body_bottom = 0.1
```

```
header = {
    'left': 0.1,
    'right': 0.1,
    'top': 0.2,
    'bottom': 0.2
}
body = {
    'left': 0.15,
    'right': 0.15,
    'top': 0.1,
    'bottom': 0.1
}
```

On the left side you have the arguments specified directly and on the right side you have them specified in form of a dictionary. It is a matter of style which one you prefer but you have both options just in case. As you will see in a moment, you can even use both ways at the same time.

There is also a special keyword with which you can set the header to body ratio:

```
hbratio = 0.2
```

Now following are a couple of examples which will be presented under the different forms in which you can enter the values.

```
im = Background('pic.png', header_left=0.2, header_right=0.2)
im = Background('pic.png', header={'left': 0.2, 'right': 0.2})
```

Here the left and right margin have the size of: `0.2 * 'width of the image'`.

```
im = Background('pic.jpg', body={'left': 0, 'right': 0})
im = Background('pic.jpg', body_left=0, body_right=0)
```

The left and right margin of the body is non-existent. They are basically the right and left margin of the image.

```
im = Background('pic.jpg', header_top=0.5, header={'bottom': 0})
im = Background('pic.jpg', header={'top': 0.5}, header_bottom=0)
im = Background('pic.jpg', header_top=0.5, header_bottom=0)
im = Background('pic.jpg', header={'top': 0.5, 'bottom': 0})
```

The top margin has half the size of the height of the header and the bottom margin is non-existent, which means that the bottom margin of the header is basically where the body begins.

```
im = Background('pic.png', body_bottom=0.5)
im = Background('pic.png', body={'bottom': 0.5})
```

The bottom margin has half the size of the height of the body.

So the values smaller than 1 are expressed relative to their respective area. Either the image width for `left` and `right` or the header/body height for `top` and `bottom`.

You can replace the values in the examples above with a number greater or equal to 1, in which case the thickness is expressed in pixels. If the value exceeds the borders of the respective area (header or body), an error will raise: `ValueError`.

The header to body ratio can be set the following way:

```
im1 = Background('pic.png', hbratio = 0.25)
im2 = Background('pic.png', hbratio = 0.5)
```

For `im1` the header will take up a quarter of the image and for `im2` the header and the body will be split in the middle of the image. Note that there is only one way to specify this keyword. You can only insert a value smaller or equal to 1.

5.4. Preview and save the image

After you have created your image it would surely be nice to preview it and if you like the result save it afterwards.

5.4.1. save_to

This is a method with which you can specify a save location that applies for all the images you create.

```
pbc.Background.save_to(*location)
```

Be aware that you will not be able to specify another save location afterwards any more.[5] If you want to change it you will have to use this method again and specify a different location. Alternatively you can enter an empty string as an argument:

```
pbc.Background.save_to('')
```

Note that there are two ways to specify the location. Either giving the path in a single string like you would do it in section 5.1 or you can comma separate the different directories.

```
pbc.Background.save_to('my/save/location/')
pbc.Background.save_to('my', 'save', 'location')
```

Both statements do the exact same thing. It is just a matter of style which you use but you have both options just in case.

[5] Refer to section 5.4.3 on the next page for more information.

5.4.2. show

This dead simple method lets you preview your image:

```
pic.show()
```

Displays this image. This method is mainly intended for debugging purposes.

On Unix platforms, this method saves the image to a temporary PPM file, and calls either the xv utility or the display utility, depending on which one can be found.

On macOS, this method saves the image to a temporary BMP file, and opens it with the native Preview application.

On Windows, it saves the image to a temporary BMP file, and uses the standard BMP display utility to show it (usually Paint).[6]

5.4.3. save

Finally you can save your image with this method. Keep in mind though that if you set a save location using the `save_to` method shown in 5.4.1 you will no longer be able to choose a specific save location for the different images with this method.

```
pic.save(name=None, *location)
```

name obviously lets you set the name under which you want to save your image. In order to specify the save location there are now two different ways of going about it.

Firstly you can prepend the path to your name which would result in something akin to what we have done in section 5.1 Open an image. Secondly you probably recognized the ***location** argument which we already encountered in section 5.4.1 so I won't go into it again. If however you use both ways, then there will be an error saying that `TOO MANY paths were given` and the saving will be skipped for that particular image.

Now let's briefly tackle the problem mentioned in section 5.4.1. If you have specified the save location with `save_to` then there are two ways you will notice the effect. If you use the first method of specifying the path described in this section, then the prepended path will simply be ignored which should be fine considering the use of `save_to`. If on the other hand you choose the second way of specifying the path using ***location**, then the error from before will raise saying that `TOO MANY paths were given`.

I will not talk about why this behaviour is occurring but either way it reminds you that you already set a *global* save location for all the images. You then should not want to alter the save location any more. A little trick to circumvent this problem though is to set the *global* save location to nothing and then setting it back to what you want. In other words, temporarily disable the global save location.

```
pbc.Background.save_to('')
pic.save('mypic.png', 'other/location')
pbc.Background.save_to('my/save/location')
```

[6] Paragraph from https://pillow.readthedocs.io/en/4.0.x/reference/Image.html?highlight=show#PIL.Image.Image.show which is the method used under the hood. Don't worry about the arguments specified on that page, they are irrelevant to us.

A. Examples of custom themes

Here are a few examples of custom themes I created. It has always been a repetitive task and sometimes quite tedious as well which is why I had the wonderful idea to make a program that does all that boring stuff for me. Keep in mind though that the following examples were not made using this program. As a side note the captions here actually show the topic of the presentation.

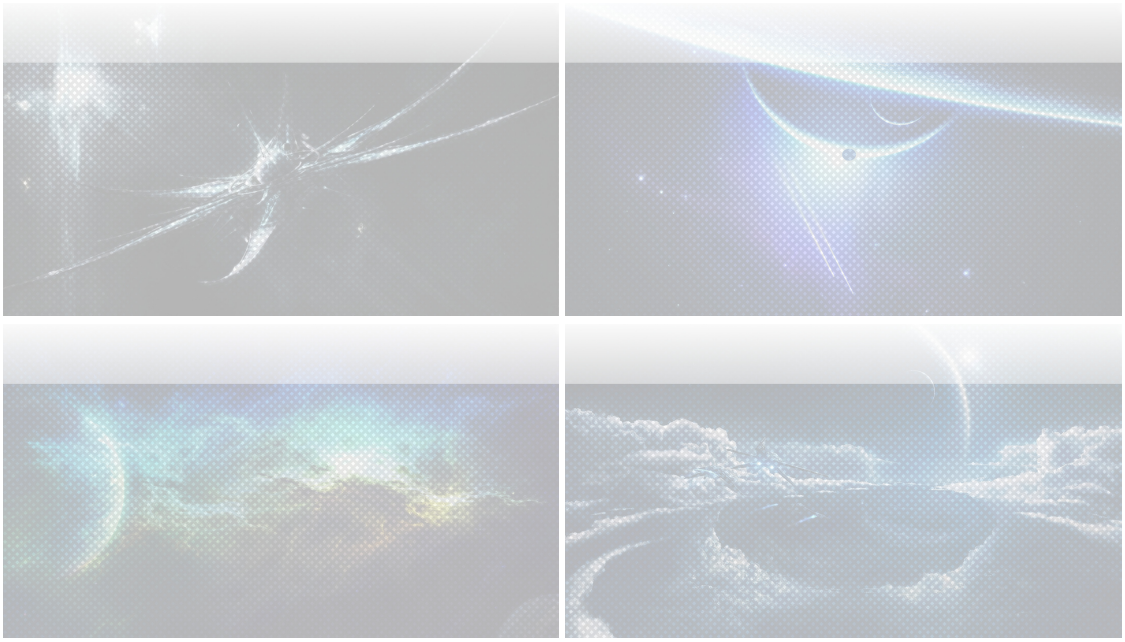

Figure 2: Where are the aliens?

As you can see in figure 2 we were talking about an alien related topic. I wanted to have a background which kind of reflects this and therefore I went for these Sci-fi designs. I added a diagonal lines pattern on top of the image to have something nice looking and lastly added a semi-transparent image on the top part of the image which defines the title area. This has all been done using Gimp.

In this presentation, seen in figure 3, we elaborated on the idea that we might not live in a *real* world but instead in a computer simulated world. The *Tron* style images seemed fitting to me as it also happens to take place inside a video game which is a simulated world. What I did here is to create a dotted pattern with Inkscape which I overlay on a blurred image. The blurring was done in Gimp just like all the overlays for the body and the header.

Lastly this was a presentation about ancient astronauts. The idea that aliens have visited Earth in the past and actually helped young civilisations develop. The backgrounds seen in figure 4 were all made with Krita. I would go as far as to say that this was the most tedious to accomplish since the amount of effects used at once made the program pretty unstable and slowed down my computer a lot. I had to close all other programs in order to keep Krita running the best it can. Firstly I created the curved

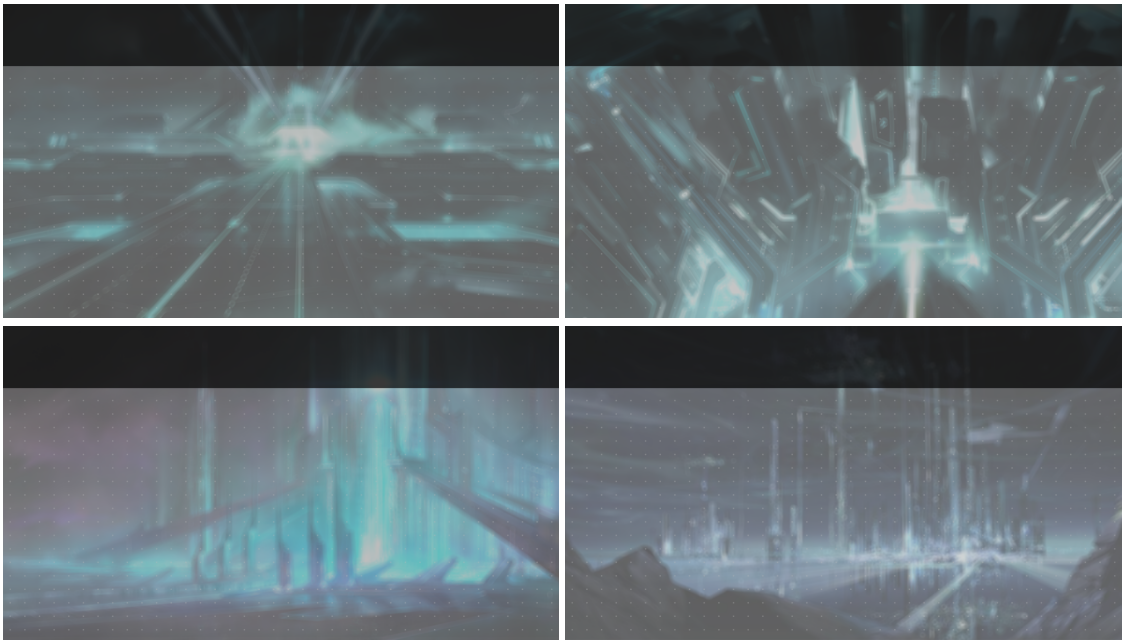

Figure 3: Are we living in a simulation?

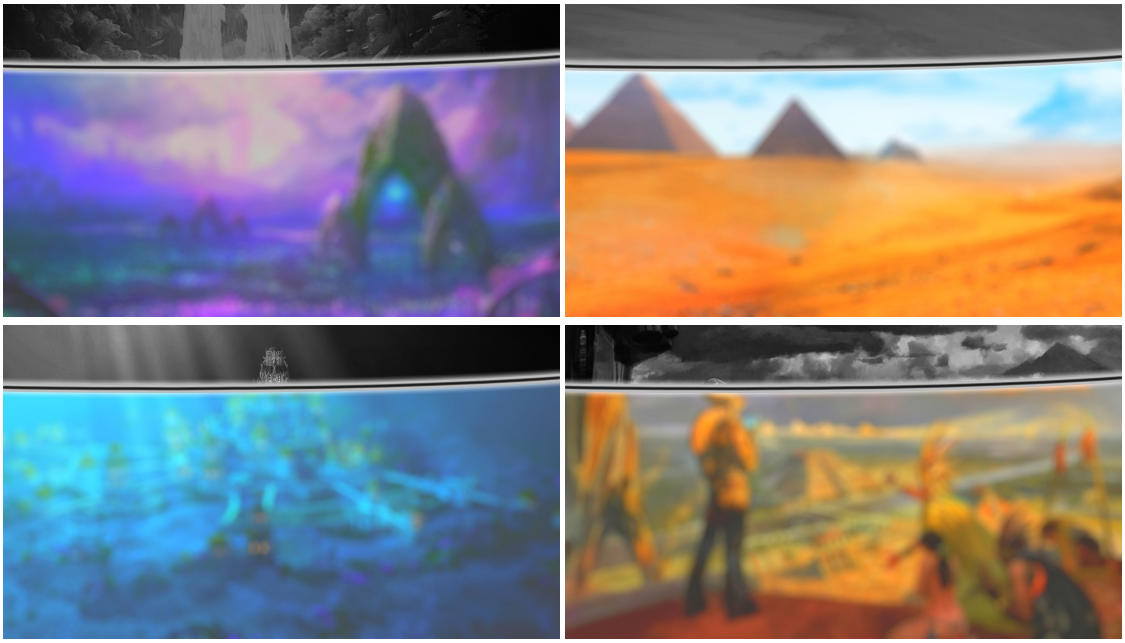

Figure 4: Ancient astronauts

line which separates the header from the body. Then I created a layer effect that blurs the image. Next I had to created a second layer effect that makes the image black and white. To combine the two I had to create a layer mask for the black and white image so it only covers the part above the separator. This case was when I most longed for a program like this one.

One imported thing that *absolutely* needs to be considered, is the readability of the text on the background. You have to make sure and make a couple of test to ensure the text is nicely readable on your background. With this in mind, you cannot use a background with plenty of black colour while using a black font colour. Especially for the images in figure 2 I spent almost a whole afternoon adjusting the lightness of the diagonal lines. If they were too dominant, you would not be able to read the text any more as they would distract and put the black text on a black background. For the images in figure 3 there is clearly too much black in order to use a black font. Using a white font is not an option either since there are pretty bright patches in the images which would cause the text to be unreadable. Thus I simply created a semi-transparent plain white overlay that sort of lightens the black colours which avoids black text on a black background. At worst we now just have black text on a light grey background which worked just well in this case.

Even though this semi-transparent overlay seems like a universal solution, which it is I believe, you have to take a moment and think about the colour it should have. A general rule of thumb I can give from my experience is to use a greyish overlay for a black font. This will cause all the black colours in the image to have that greyish colour. I know that technically this isn't very correct but for the sake of explanation I will just leave it at that.

In case you wonder, these three presentations were very well receipted. On multiple occasions people asked me how in the world I made those backgrounds and how I managed to make them change from slide to slide. It is actually not all that complicated once you know how to do it (duh), and this program will hopefully eliminate the process of hand crafting the various backgrounds, which are all made with the exact same procedure, multiple times.