Using a processor for memory -efficient GUI display driving

Master Thesis
NTNU
Trondheim, Spring 2022

Candidate (Surname, Name): Saure, Runar				
Date:	**SUBJECT:**	**GROUP (name/nr):**	**PAGES/APDX:**	**BIBL. NR:**
22.06.22	TFE4930	None	86 / 8	N/A
SUPERVISOR(S): Glenn Ruben Bakke, Snorre Aunet				
TITLE: Using a processor for memory-efficient GUI display driving				

Abstract:

While microcontrollers are getting stronger by the day and more advanced technologies are developed for embedded systems, it seems logical to use these in more complex embedded systems such as a human-machine graphical interface. However, cutting costs is always a priority, and by using techniques to limit the RAM-usage and CPU Processing power required through the use of a line-based description of the graphical scene, it seems possible to better optimize the system for lower cost microcontrollers. The implementation of a high-level model based on this idea was tested, and showed mixed results. The RAM usage was noticeably better, ranging from 26 785% and upwards in saved RAM space, when not considering compression. The runtime was however noticeably worse, struggling to perform the same as a reference model. Given the basis of this thesis, it is possible to further optimize the system, such as by using a lower level model to better match the runtime of a standard render model.

Contents

1 Introduction - Description of Task

This thesis covers the research of a line-based render definition on a system driving a graphical interface through a human-machine interface, for example, a screen. It will cover the exploration of increasing efficiency of memory and processor usage, to accompany slower low-power GUI-driving microcontrollers. The Line-based implementation will be compared to a reference model using more standard methods of rendering solutions more similar to standard implementation in double-buffered systems. Both the complexity and timing between such systems will be compared to create a foundation to more mathematically find a budget for each line in the line-buffer to compare to the standard rendering reference. This study can be considered to be split into multiple smaller parts, consisting of analyzing the RAM usage for each individual solution given different scene compositions, in addition to analyzing the time budget for different operations to give a generalized idea of what operations can be ran within a given line before it exceeds the reference models timings. The project is based on a high-level implementation, utilizing Python as the coding language for all functions.

This thesis covers multiple sub-elements to make the line-based solution possible, such as a specialized scene descriptor to construct the graphical elements and handle any operations performed on it. It will cover the general principle of Colour Lookup Table, alpha blending, and masking operations, RAM, and how these are implemented in the system. Analysis of the two systems, and comparisons will be covered.

This thesis is based on previous work on a pre-project report studying the basis of the task, with the focus moving towards a deeper research of underlying theories and implementations of the previous task. Some of the subjects covered by the previous work will be covered and expanded upon in the theory part, but reading the previous report is recommended to get a better overview, and is included in the appendix: [Pre-project report for Master Thesis].

2 Background - Theories

2.1 RAM

Most MCUs and larger systems contain what is called “Random Access Memory”, or RAM for short. This is a volatile memory unit, used for intermediate storage of data as a mid-level between the CPUs cache, and higher storage units such as Flash, EEPROM, SSD etc. This implies that any data too large, or not important enough for the cache is placed in RAM. One such case is when performing image operations in a system, where the temporary collected data, for example when working with the whole frame of a picture, is stored in RAM for fast access, while any sub-pixels currently being operated reside in the CPU cache. However, as the RAM is volatile, any data will disappear during any power loss, which means that storing critical data long term in it is considered a bad idea.

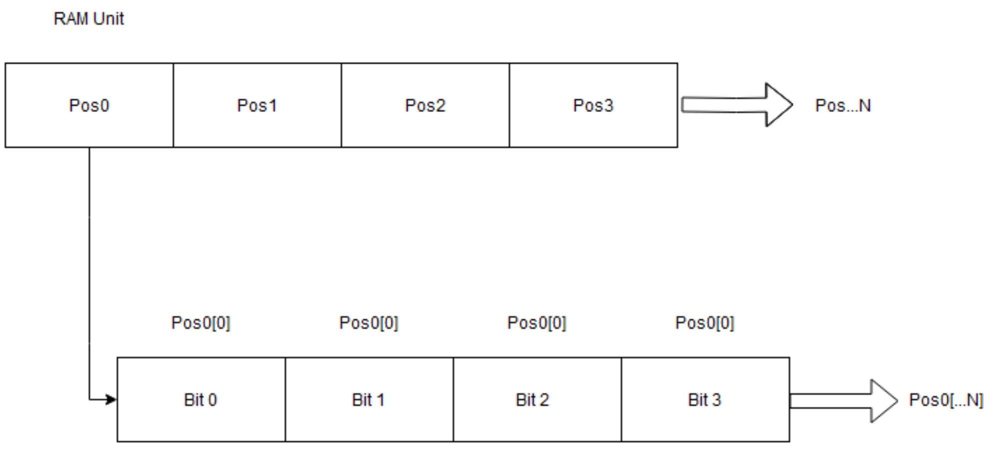


Figure 1: RAM array visualized

To better understand the inner workings of a RAM-module, refer to figure 1. The RAM is represented as a long array of size N, with a subset-array of arrays given size [N], which usually is set to 8 bits. Each of these will act like a register, having their own address under the parent array, but usually only one data-byte is stored in each RAM Pos-unit. It is possible to split the array into a high- and low nibble of different data sets, but is not recommended to do manually. Usually, a larger data-set will be split out over multiple sub-arrays, such that the total data is stored in the range of Address to Address + N. When the data is to be collected, one uses the allocated address as an argument into the RAM, which then collects the required byte, or puts any data into the supplied address if performing a write. In more common systems, it is possible for the data to be spatially placed, however, in this project it is limited to simulating a sequential placement.

2.2 Colour Lookup Table

In order to represent a picture, a Colour Lookup Table (Hereby known as CLUT) is needed for the picture, unless a global CLUT is provided by the scene descriptor. By providing one of these, it is possible to both display and modifies the colour characteristics of the provided picture. In the scene descriptor for the current model, the CLUT is implicitly imported when fetching the picture by setting the picture pixel-points RGB-values directly and using standardized CLUT-values to represent the picture as-was, which usually is represented as 24-bit images. This way, any stored picture with a normalized CLUT will look as before it was imported. However, in the case that the user wants to modify the picture with a given hue, or a set filter, it is possible to change out the Picture's CLUT or apply a global CLUT to modify the picture, for example by giving it a more red tint overall.

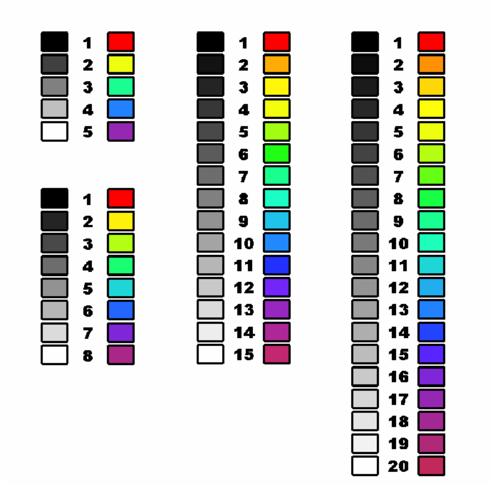


Figure 2: 1D CLUT. [1]

The way the implementation of CLUT works in the current code acts as an expansion, or a reinterpretation of the basic idea of CLUT, allowing greater control of each colour range, over for example a simple 5-bit shared CLUT, which would limit the colour space to up to 10 unique colours, where usually a selection of bit values corresponds to set colours in the RGB or CMYK range, and a control bit to select if the value should correspond to a grey-scale mapping. Multiple interpretations of such colour spaces can be seen in Figure 2, showing multiple bit-sizes and solutions. In the 2-dimensional model, the extension of the colour range to the colouration of each channel corresponds to "completely absent" at 0, to "fully opaque" at 255. At smaller sizes, this extension might start looking close to the basic idea of CLUTs, however, the 1-dimensionality of the basic concept will always be more space-saving than any higher-order array, but at the obvious cost of precision.

The CLUT is based on a 2D implementation of size 3*(Bits), where Bits imply the size of each R, G and B colour palette. To better visualize how this looks, Figure 3 is included, where each shade of a colour X and a colour Y is incremented in the X/Y axis for each sub-element. Each jump between squares symbolises an increment of colour Z, given as an offset of the size of the two previous colour channels, and shows how increasing the Z-colour factor affects the colours for the previous channels. Together, these three will create a specific colour given by their respective input R, G and B values. A more complex representation could be made by stacking

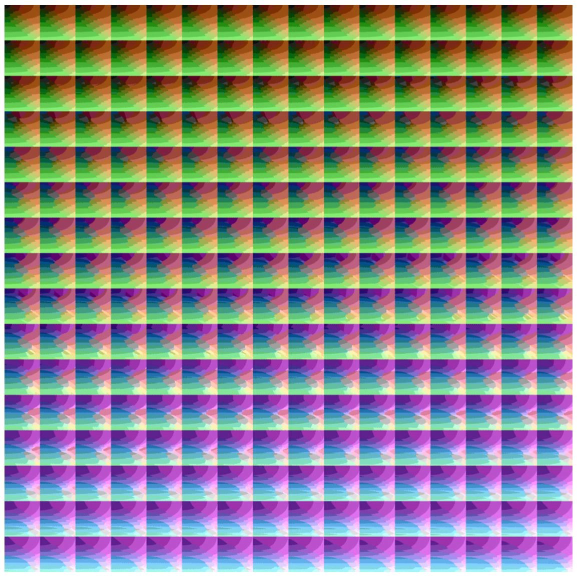


Figure 3: 2D CLUT [2]

the Z-axis of the previously created 2D palette visualization, creating a 3D cube visualization. This representation could make it even easier to visualize how each R, G and B increment or decrement affects the output colour value. A single "point" in the cube will then result in the colouration of the current pixel on the display. Given an equal size of each channel, a cube of the size $(n)^3$ can be generated from a 2D CLUT. One such example is shown in Figure 4.

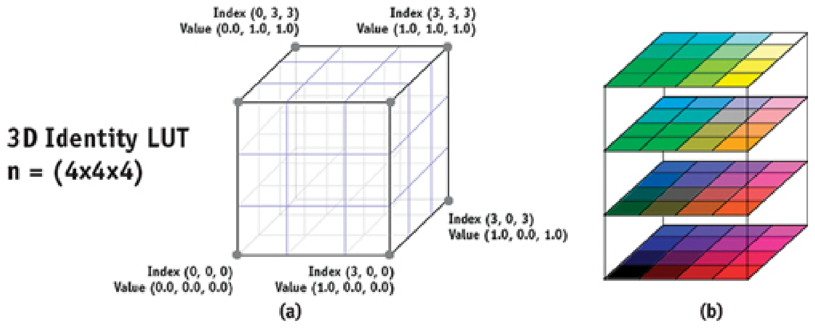


Figure 4: 3D visualization of 2D CLUT. [3]

2.3 Alpha Blending

Alpha Blending is the process of combining two layers of overlapping pictures with the appearance of transparency, to make an illusion that the background shines through onto the overlaying picture. This is done with a fourth positional bit for every pixel, bumping the bit count up to, regularly, 32-bit in images. Using this extra byte, often referred to as "Alpha value", allows one to apply an algorithm that weights said alpha as a transparency factor. This method uses said alpha to combine the foreground with a "matte", which often is a background picture, to multiply the source pixel with the alpha to create an output pixel, which when combined with every other pixel is referred to as the composite. Usually, the alpha range is mapped to values between 0.0 and 1.0 such that 0 equals a fully transparent picture, where the foreground disappears and the background is rendered, while an alpha value of 1.1 creates an opaque foreground picture. This method is usually referred to as an “Over” operator, although more complex operators exist such as “in”, “out”, “atop” and “xor”, but these go outside the scope of the thesis, and is recommended to get to delve on your own.

To achieve an "over" alpha blend, a standard, easy-to-run formula is usually used, given by

$$Composite = Background * (1.0 - Alpha) + Foreground * (Alpha) \tag{1}$$

which is applied per pixel, per colour channel, and outputs a blended byte as a final composite of the foreground and background picture. Here, the Foreground will be the overlaying picture, and the composite will be the end result picture of the operation. One thing to note, which makes this effect possible, is how the background decreases in value as the alpha is increased, which in turn increases the factor of the foreground.

Unlike the R, G, and B channels, the alpha value is most often used as a per-picture factor, where the picture can be seen as “Equally transparent”, instead of as a per-pixel factor, where the picture would get a fading effect, although the latter is fully possible. As a real-life example of how alpha blending works, one could imagine holding a red see-through textile in front of an object and seeing through the textile. To help visualize further, refer to Figure 5

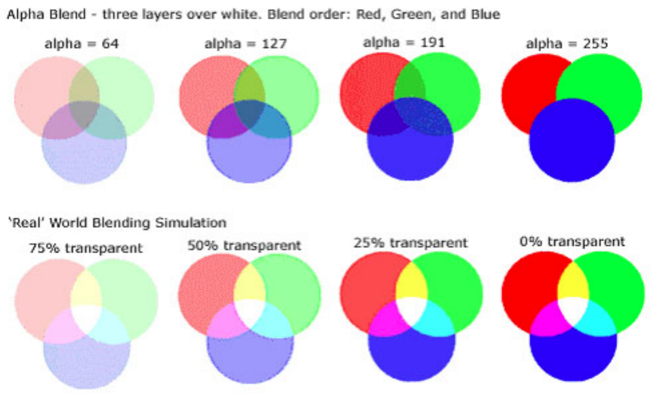

Alpha Blend - three layers over white. Blend order: Red, Green, and Blue
alpha = 64
alpha = 127
alpha = 191
alpha = 255
'Real' World Blending Simulation
75% transparent
50% transparent
25% transparent
0% transparent

Figure 5: Examples of blending at various alphas. [4]

If the process of alpha blending seems curious, it is possible to read more about the process in an article published by Sudipta Maji and Asoke Nath [5]

2.4 Masking

The effect of using masking on pictures allows one to do simple “cut-outs” of a picture, often defined by an attached stencil-like picture, referred to as the “mask”, or “bit mask”. This mask file is composed of a picture-like format, where each bit is defined as 1 or 0. This means that masks usually are small, which could make them a powerful tool for applying graphical effects on weaker hardware. This effect is often used in graphics where a certain part of a picture is intended to overlap a background picture, but the intended picture is not squared, or just a part of a larger picture. By using the bitmask, the relevant part of the picture is cut out of its original file, and placed over the background image, resulting in an output picture consisting of the new compound image.

2.5 Horizontal Sync

Horizontal Sync, or h-sync for short, is a signal that is given at fixed intervals to the monitor to change the current drawing line from current Y to the next Y. This is done at the time of the screen resolution in vertical orientation, times the target frame rate. Given a 24 frame rate target, which is categorized as the preferred frame rate in a study published by Farid Pazhoohi and Alan Kingstone [6], and at a 600 vertical resolution, it would approach a time limit of $6,94 * 10^-5$ seconds, or 0,00694 millisecond time limit per horizontal line before it changes line. By using the formula

$$H - sync = \frac{1}{(FramesPerSecond) * (Vertical\ length)} \tag{2}$$

It is possible to find the time frame given for each line operation, and will be harder to reach if the frame rate, horizontal or vertical length is increased, which needs to be carefully considered and balanced when compared to the complexity of the scene.

2.6 Frame Buffering

In most modern GPUs, it is not unlikely to find what is called a frame buffer. This buffer is a dedicated portion of the GPUs RAM specialized in containing any currently rendered frame represented as a bitmap, which is ready to be output to any external I/O devices. This could be items such as a screen, another buffer etc, delivering a GUI item, such as graphics, texts, or similar rendered products. GPUs contain at least one rendered picture and are sent to the external I/O device as soon as a “get” signal arrives. As soon as the rendered frame is gotten, the buffer is usually flushed, and a new frame can be stored. This method might employ some tight time limits, all depending on the current application and complexity of the rendered scene, which, in the worst case, results in a bad user experience or hiccups to external systems that require the rendered frame within the given limits. The size of the frame buffer might vary, but is at a minimum the required data size of a full picture of the maximum supported resolution.

2.7 Double Buffering

Some GPUs might have only one buffer, but implementing multiple buffers has become common among hardware developers. The main drive for using multiple buffers is the most likely removal of any stuttering of the picture flow, flickering of the picture between the time a buffer is flushed and a new picture is read, or tearing of the picture, the GPU is out of sync and draws the next frame at a higher rate than the screen, and parts of the old image remain while the new one is drawn over it per-line. These are handled by allowing the next frame to be rendered onto the secondary buffer, while the primary buffer keeps displaying the current data set. this way, a refresh of the buffer does not happen, and the screen remains active with the current picture, instead of producing tearing or flickering. When the next rendered frame is ready, a MUX changes the output status from one buffer to the other, switching the primary- and secondary status of the two buffers, allowing the previous primary buffer to refresh and receive the next frame. This method gives some extra slack for the GPU, allowing for some smaller delays at heavier loads without compromising too much of the user experience.

A drawback of the double buffering method, is that the amount of buffer memory must double. Some implement this at hardware-level, meaning a larger footprint on the PCB, while others might go for a software solution, by using, for example, a portion of the RAM for the buffer data. This becomes a problem with larger pictures quickly occupying up the space of valuable RAM , especially when considering smaller MCUs which have limited RAM space. Given that the rendering often apply multiple operations on a frame before sending it out, it will need to temporary store the full frames of the picture, and especially when multiple pictures are needed for the modification, the RAM usage might give large spices that leaves less resources available for the rest of the system, which is unwanted.

2.8 Line Buffer

As the problem of a double buffer potentially eating up too much RAM space during intense operations which could hamper the performance of the rest of the system, a more conservative approach might be necessary. This is where the idea of reducing the required buffer size comes in, by utilizing a reduced footprint through rendering only one line at a time, which greatly improves memory usage, but comes at its own compromises. The line buffer is allocated in the ram, which only increases in footprint by the vertical length of the frame,

RAM timing could also be influenced, as in total, more RAM gets and puts are called, which, depending on the implementation of the RAM, could create some increase in run-time when compared to the reference model.

2.9 Scene Descriptor

To help the process of knowing what data to get when, a scene descriptor format is necessary to adapt to the proposed line-based rendering technique, such that the system collects the smallest, correct amount of data, and thereby occupies the least possible RAM space. The Scene Descriptor will keep data about the picture, such as which operations to apply at each scene, how the layering of multiple elements is set up, and any additional references, such as a pointer to the mask and CLUT to be used.

By using the scene descriptor during rendering, the rendering algorithm will allow the scene to be constructed in the most efficient way, and can also easily swap the scene operations on-the-fly, given that something injects the information into the descriptor. The next time the scene is constructed after such operations, the displayed picture on a screen will have changed, given that the new operation sets are within the h-sync timings given by the hardware.

2.10 Big O – Complexity

Big O-notation is a common way in computer science to measure the complexity of a given algorithm, to check its efficiency in an easy and simplified way. By using the steps through the method, we can find a value of how the complexity of the algorithm varies with the input size N, without being machine-dependent. The way a Big-O analysis is performed, is by going through the algorithm step wise, in simple steps the same way a computer would interpret it.

When analyzing an algorithm, there most common ways of finding a measurement of it is by finding either its best case, average case, or worst case factor. Big-O covers the worst-case, which is often one of the most valuable feedback to get, as it tells the designer something about the most time-critical component.

There are some simple considerations to remember when doing a Big-O analysis. First of all, some terms will dominate the other terms, because the term simply is of a higher order, and any lesser terms will become irrelevant in the larger scheme. The following order is considered when doing an analysis:

$$O(n!) > O(2^n) > O(n^2) > O(n * logn) > O(n) > O(log * n) > O(1) \tag{3}$$

Any operator more left biased than the others found when analyzing the algorithm, will then dominate and remove the lower order terms. Any constant, such as 3n, will also be reduced to simply O(n) as the size of n rises, because of how the n dominates the constant factor.

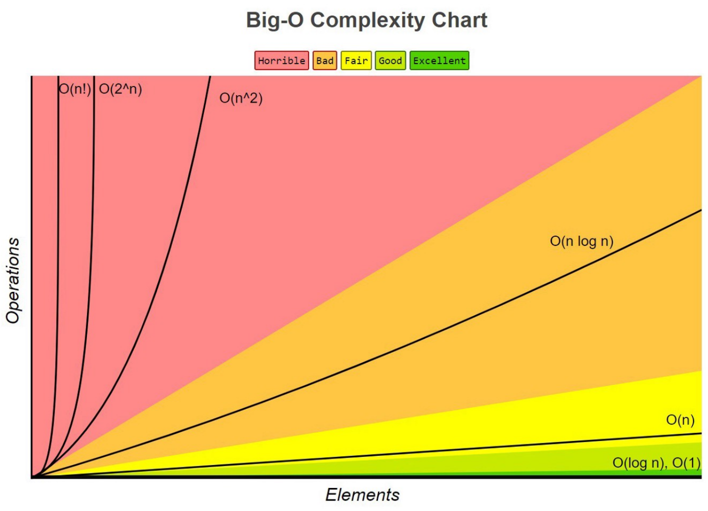


Figure 6: Big-O Complexity. Collected from [7]

Figure 6 show how each term affects the complexity, and thereby the run time of the scene, given by the numer of elements n that are put into the algorithm. As an example, would the line buffer algorithm be of a factor of O(2^n), a larger array would quickly impact the system, eating away both the RAM and run time at record speeds when increasing.

2.11 Python Packages

During the project, three external libraries were used for some operations, and is necessary to include for the code to work, as they are used in important parts of the code, to simplify some of the process during the design. These libraries can be replaced in later iterations, if it is deemed worth it to do so.

2.11.1 PIL - Python Image Library

To import and export the pictures from and to a .bmp format, and to quickly change the picture mode from RGB to RGBA at certain stages, PIL's Image sub-library was used. This allowed for easy handling of the I/O part of the images, removing Unnecessary time spent on building additional libraries that already exist.

2.11.2 Numpy

Numpy was included and often used in the process of transcribing the imported images over to an array-format, to easier handle them in the operation sets of the task.

2.11.3 time

time was included for measurement, as to find the real-world run time of the operation sets.

2.11.4 matplotlib - pyplot

A simple library that allows for plotting operations and displaying. Used to create the histograms.

3 Methodology

3.1 CLUT Implementation

The implementation currently supports a given input argument for the colour space to generate an empty CLUT of a given size, and a function to input any user-defined CLUT into the newly generated shell. This allows the user to either modify the picture to a set requirement or to experiment with colours to see how one can affect a stored picture. To recreate the function used in the model, it is recommended to use the following algorithm, with adjustments where necessary:

Algorithm 1 CLUT Apply Algorithm

```
1: procedure APPLY CLUT($Picture, newCLUT, Offset$)
2:     Function Initialization
3:     Store Picture Data in RAM
4:     for Length (Picture) do
5:         if $Pixel = Unmodified$ then
6:             Collect current R, G, B values at current x + offset
7:             Find and collect corresponding value in New CLUT
8:             Input new values into corresponding position
9:             if $Layer \neq 0$ then
10:                Set current pixel to modified
       Return Modified Picture Line
```

Running this algorithm should produce a colour-filtered picture, matching the R, G and B channels to the new CLUT. If this is to be implemented to support variable CLUT sizes, modifications are necessary to get it to run. The functions "GenerateTestCLUT" and "ChangeCLUT" were also made, but are not of importance to the model. They can still be found included in the CLUT-file in the appendix.

3.2 Alpha Blending Implementation

The Alpha Blending assumes that the foreground picture has an RGBA-format, and has a valid value for the Alpha-channel. The current implementation uses a single alpha for sampling of the whole picture to reduce the amount of additional operations, but to allow for fading, simply fetch the current alpha channel value from the stored line at the current active pixel, and apply it pixel-wise.

Algorithm 2 Alpha Apply Algorithm

```
1: procedure Apply Alpha($Foreground, Offset, Background$)
2:     Function Initialization
3:     Store Picture Data in RAM
4:     for Length(Picture) do
5:         if $Pixel = Unmodified$ then
6:             Get pixel at current X + Offset
7:             Apply Alpha to each channel R, G, B, using channel A
8:             Use mathematical Alpha Blending formula
9:             Store blended data
10:            if $Layer \neq 0$ then
11:                Set current pixel to modified
12:            Return alpha blended picture Line
```

3.3 Masking Implementation

To apply a masking operation, the steps are pretty straight forward.

Algorithm 3 Masking Algorithm

```
1: procedure Apply Mask($Foreground, Offset, Background, Mask$)
2:     Function Initialization
3:     Store Picture Data in RAM
4:     for Length(Foreground) do
5:         if $Pixel = Unmodified$ then
6:             if $MaskBit(x) = 0$ then
7:                 Pass through background pixel
8:         if $else$ then
9:             Set output pixel to Foreground
10:            if $Layer \neq 0$ then
11:                Set current pixel to modified
       Return Masked picture Line
```

3.4 RAM Implementation

Two functions are central when implementing the RAM, in order to store and access any data residing in it. in addition, a class is made, simply called "RAM", which contains three elements:

The array size data of the RAM, which creates space for the number of entities to be stored. Each entity space the array has an undefined size, as the

Note that a put_specific function is also included in the code, but uses the same placement logic as the regular put, but instead of a for-loop, uses an extra input-argument, "Address", to go directly to the given address to put the data. This function is usually used to overwrite existing data, such as a picture array that has been modified for other layers to use.

Algorithm 4 RAM put function

```
1: procedure RAM PUT($Data$)
2:     for Length(RAM Array) do
3:         if $Arrayplacement \neq occupied$ then
4:             Put data into place
5:             mark placement as occupied
6:     Return address
```

When data is stored in RAM, it is natural to also collect the data when needed. This is done through the RAM get function. For any function where a picture line is referenced to, and RAM is used, replace the variable name with the RAM Get function to fetch and operate on the cached picture line.

Algorithm 5 RAM get function

```
1: procedure RAM GET($Adress$)
2:     if $Addresscontainsanydata = True$ then
3:         Return data
4:     if $Addresscontainsanydata \neq True$ then
5:         Return error
```

As visible from the algorithm, it is important to store the address returned from the put function in a temporary variable in order to access it at a later time. Foregoing this will result in values getting lost, the RAM getting temporary cluttered, and generally a lot of thrown errors for any functions used.

As to avoid a quickly filled RAM as time passes and functions are ran, an additional function is necessary in order to remove the data from the array position, such that new data can be stored. This is handled by a function called clear, with a general implementation shown below.

Algorithm 6 RAM clear function

```
1: procedure RAM CLEAR($Adress$)
2:     if $address = "All"$ then
3:         for Length(RAM Array) do
4:             Clean the data for the current array entity
5:             mark the space as clear
6:     if $Else$ then
7:         set current array entity to empty
8:         remove any data stored
```

3.5 Scene Descriptor Implementation

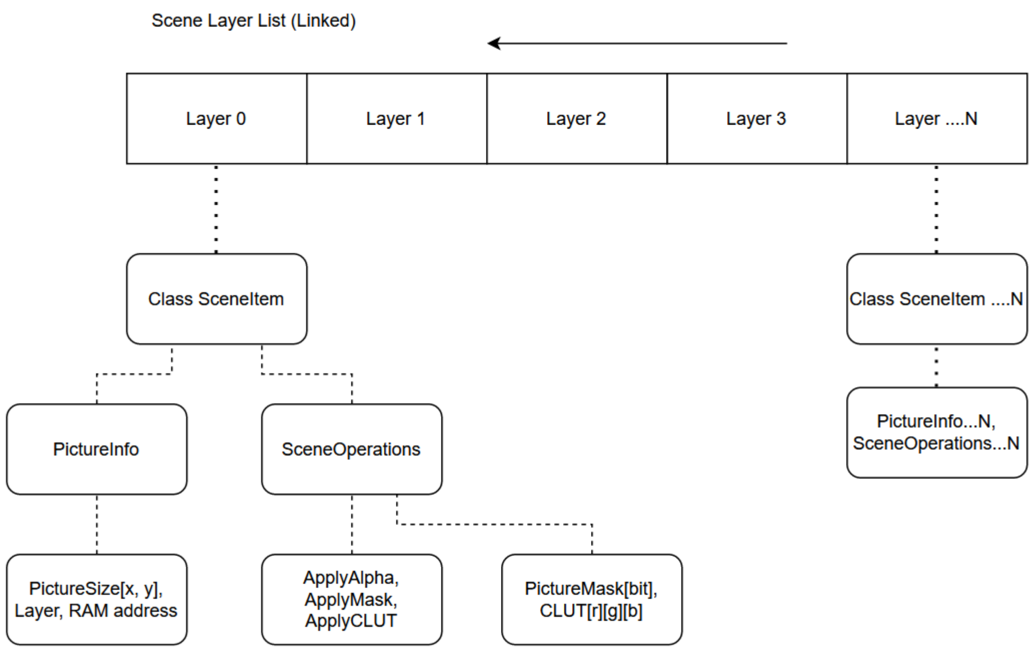

Scene Layer List (Linked)
Layer 0
Layer 1
Layer 2
Layer 3
Layer ....N
Class SceneItem
Class SceneItem ....N
PictureInfo
SceneOperations
PictureInfo...N, SceneOperations...N
PictureSize[x, y], Layer, RAM address
ApplyAlpha, ApplyMask, ApplyCLUT
PictureMask[bit], CLUT[r][g][b]

Figure 7: Model for Scene Descriptor

The Scene Descriptor, in addition to the Scene List, as shown in Fig 7, is one of the most important parts of the model that allows for a controlled line-based rendering technique given a line-limited buffer. It acts as a wrapper around a set of pictures, and allows for manipulation of commands, placement, modifications and similar on a per-layer level. As each Scene item is initialized and loaded, it is placed into a Scene Layer List, which handles the rendering priority of the scene. As the current implementation works on a top-down rendering orientation, it is necessary to consider this when building the scene. To build the scene, it is simply placed as an argument in the render() function, where it works its way down, modifying and placing the scene given the commands of the wrapper. As an example of how the final output using the scene descriptor looks like, refer to Fig. 28, 29 and 30. As an example, Scene 3, Fig 30 was built by using the components of Fig. 35, an input CLUT, and two text pictures with masks. By working its way topwards-down, each element was modified according to the layers description, considering the lower layers, and built the scene line by line and sent it to the line-buffer, which again sends it to the screen, which created Scene 3 with high artistic value.

3.6 Rendering Structure Implementation

To actually take use of the scene descriptor, a rendering function is necessary to unpack and build the output picture from the input layout. To support this, the Render() function was built, which works its way through a Scene Layer list, given by the current scene to be displayed, and does the necessary operations to display the picture they way the designer thought it out. The Render() imports the said Scene Layer list, and starts by rendering the highest layer. If any operations are performed on it, it will mark the area affected as "used" Through a small 1-bit buffer of size Length(X_Screen), which is used by lower layers to check for "keep-out" areas by marking the affected pixels as "dirty". As the top layer finishes its marked operations, it switches down through the next layers iteratively, and perform their operations, kept out by any "Dirty Pixel", until the bottom layer is reached. This layer is the Background layer, which has a more limited scope in functions, as it has no lower layers to affect. After performing or bypassing a CLUT-option, it fills in any "Clean Pixel" locations that are left on the line with the background data, and then returns the scene line out to the buffer.

Algorithm 7 Rendering Function

```
Create scene
procedure RENDER($Scene\_list, CurrentY, RAM$)
    Initialize function
    create output Line buffer in RAM
    Start at top layer
    Save picture line in RAM
    if Current layer active in line CurrentY then
        if ApplyCLUT == True then
            input relevant layer info into CLUT function
            for Length(Picture) do
                if Pixel modified then
                    move modified pixels into Line buffer
        if ApplyMask == True then
            Input relevant layer info into Mask function
            for Length(Picture) do
                if Pixel modified then
                    move modified pixels into Line buffer
        if ApplyAlpha == True then
            Input relevant layer info into Alpha function
            for Length(Picture) do
                if Pixel modified then
                    move modified pixels into Line buffer
        if Background layer reached then
            Fill free/missing pixels from buffer with background
            for range(Line) do
                if Pixel modified == False then
                    save background pixel to line buffer pixel
        Move to the next lower layer in Scene List
```

4 Results

4.1 A note about results

Due to the amount of result pictures, most of the lesser relevant ones have been moved to Appendix A.

4.2 Per-line timing diagram

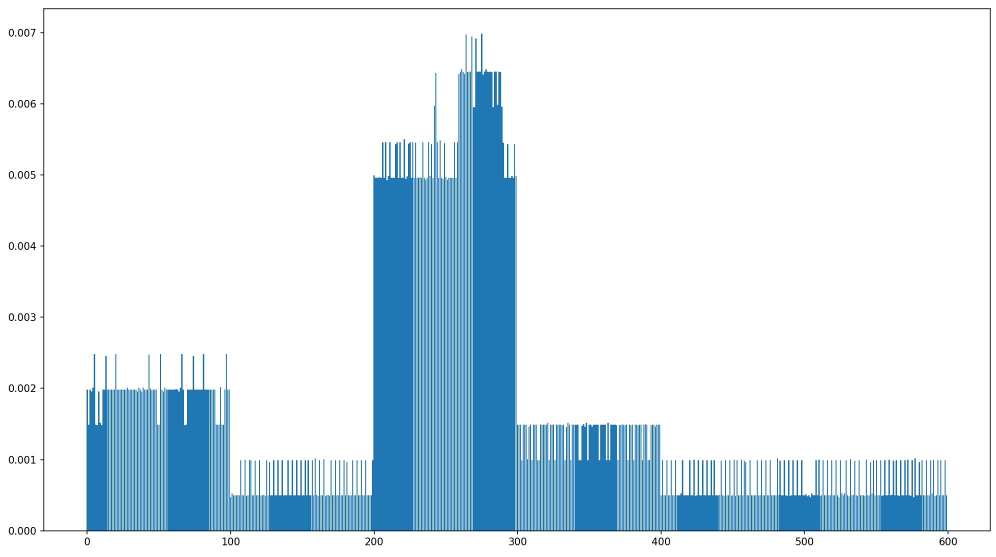


Figure 8: Time usage Scene 1

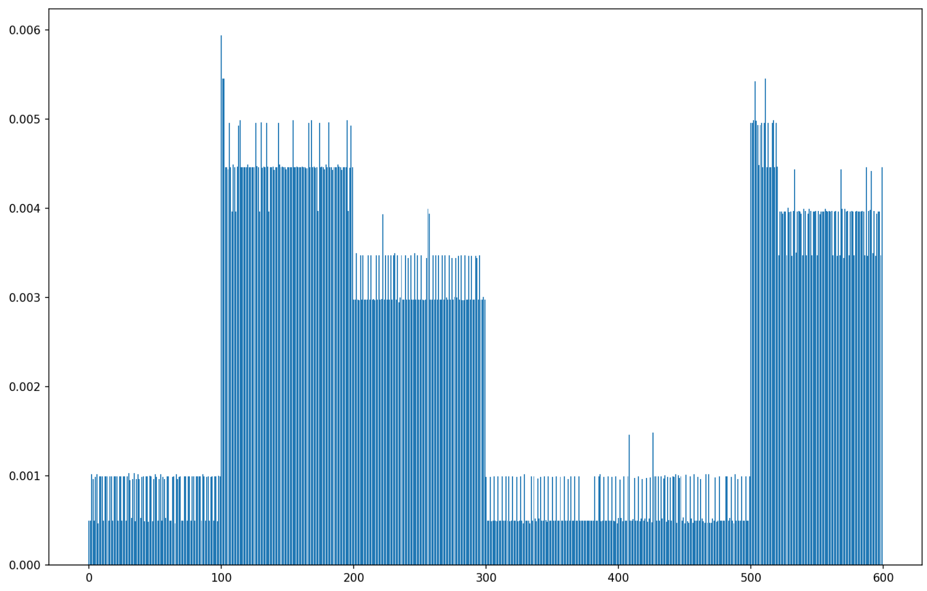


Figure 9: Time usage Scene 2

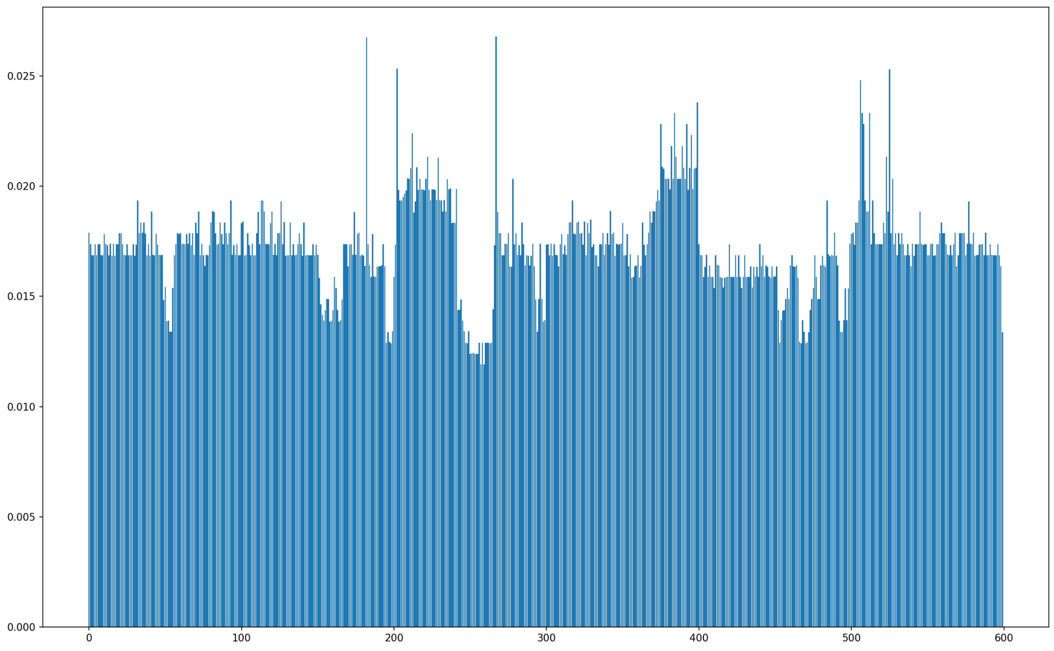


Figure 10: Time usage Scene 3

4.3 Reference Scene Timing

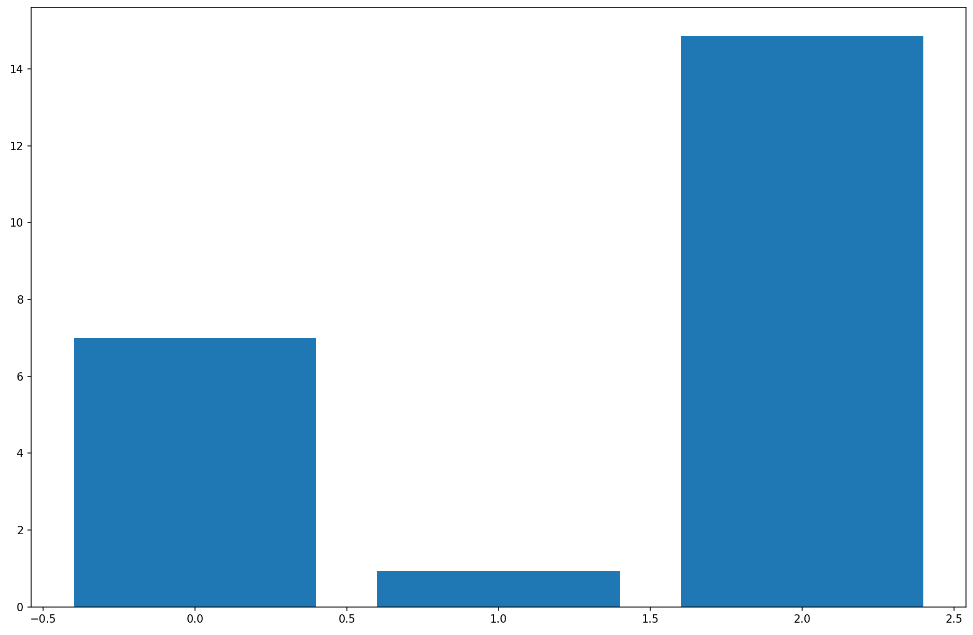


Figure 11: Runtime with Reference model on Scene 1, 2, 3

4.4 RAM-usage per line

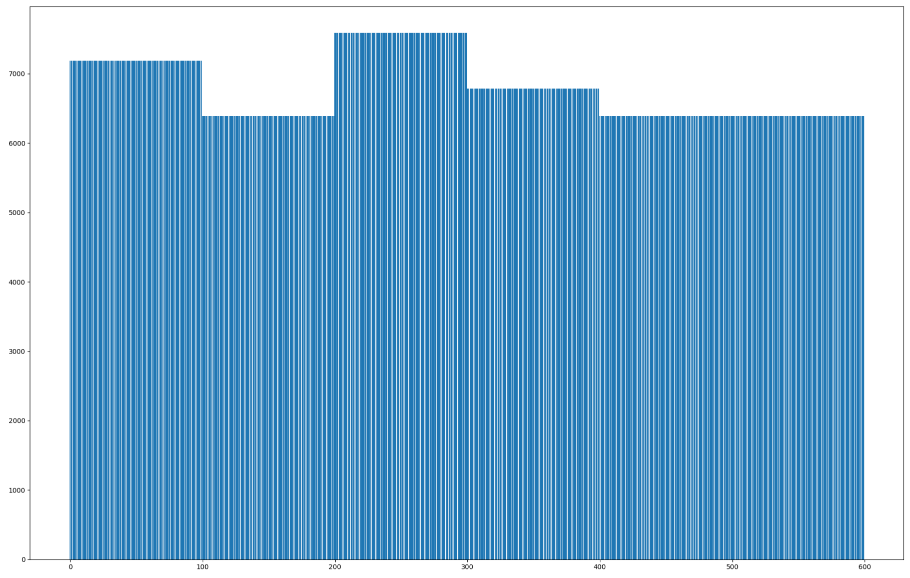


Figure 12: RAM Usage scene 1

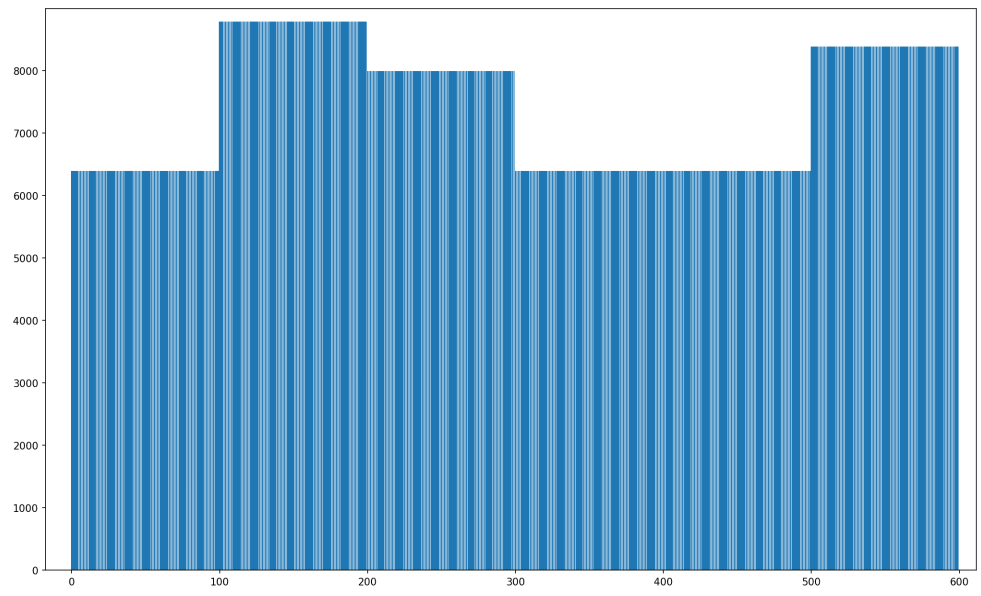


Figure 13: RAM Usage scene 2

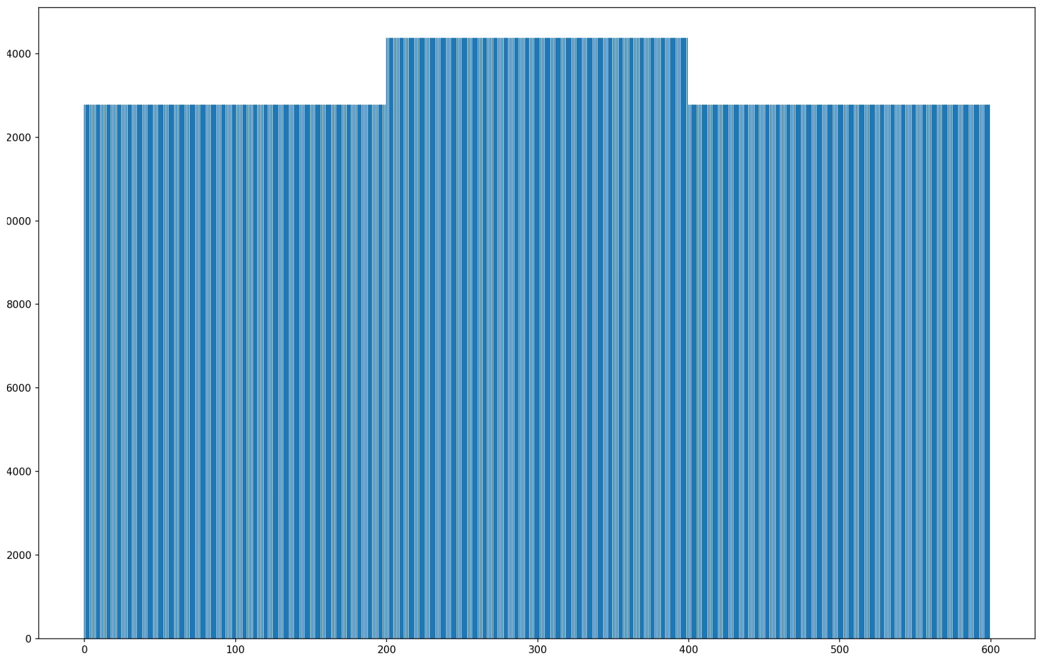


Figure 14: RAM Usage scene 3

4.5 Relative run-times

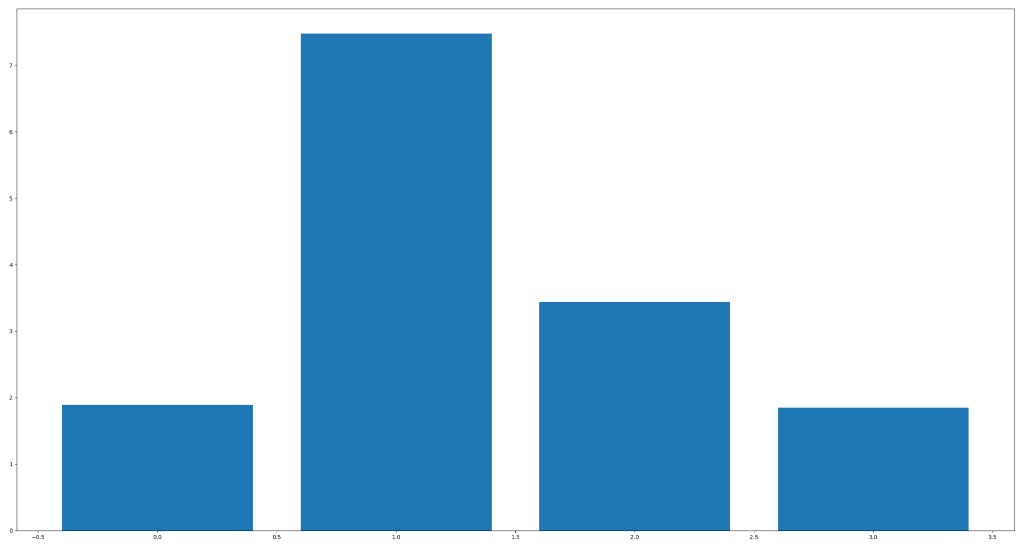
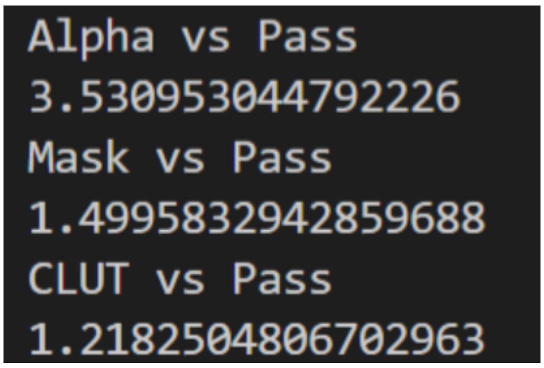


Figure 15: Time spent in each operation, given the same picture



```
Alpha vs Pass
3.530953044792226
Mask vs Pass
1.4995832942859688
CLUT vs Pass
1.2182504806702963
```

Figure 16: Ratios between operations and passing through a simple picture

4.6 Output Scenes

Refer to Appendix A Section 3

5 Discussion

Considering the resulting output graphs from Fig.8, Fig. 9 and Fig. 10, which give the execution time in a per-line fashion, it is possible to see certain trends when compared to the implemented reference model, with the timings shown in Fig. 11. The exact values of the reference model, at exactly 7.02, 0.93 and 14.49 seconds can be divided to its Y-line lengths to find the comparable render time on an average-per-line basis, which would correspond to the actual run-time limits of each line, comes in at (1) 0,0117, (2) 0,00155 and (3) 0,02415 seconds. These are "at slowest" estimates, meaning that no function can run slower than this time in order to perform at the same rate of the reference model. Considering Scene 1 and Scene 2 compared to the reference model, it seems like the implementation runs slower than the reference for the given scene. Many factor can affect this, including how the run time compiler interprets each of the models, the higher number of RAM accesses possibly slowing down operations more frequently on the suggested model, and general optimization issues with the line based implementation due to designer choices.

Scene 3 is an interesting case however, being the most complex scene of all three. The timing diagrams of scene 3 imply that the line based implementation will reach its goal at over 90% of the lines, meaning that any factor making the reference rendering quicker at scene 1 and 2 is lost when constructing a more complex scene. One factor that could affect this, is the RAM put and get commands have to be called more times for the complex scene in the reference model, which means storing, getting and clearing the large pictures at a much higher rate, finally impacting the run-time in such a way that the line-based rendering is more viable.

A RAM comparison is also in place, showing the current amount of stored bytes for Scene112, Scene213 and Scene314. The reference model is quite forward to calculate, adding the total size of the two pictures together, as both needs to be cached in RAM at the same time. Given the 800x600@24BPP used in the task, the minimum size of the background buffer will always be at 1 875 KB when stored directly in RAM at full size, in addition to overlaying pictures used at the same time, which given the constructed scenes, vary from. This already overshoots the worst-case value of the Line-based buffer, coming in at 7,8125 kilobytes. This implies that for a RAM-limited micro-controller unit, a line-based renderer is preferable over the reference model, as it has a reduction of 26 785%, which is very highly viable for small micro-controllers.

To minimize any variations, it is better to explore the comparison in a complexity and Big-O notation . Starting with a Big-O notations, a quick analysis shows that all are of a general same form, with slight variations.

The active part of the Alpha can be simplified to

```
procedure ALPHA(SomeInput)
    Function Initialization
    Store Picture Data in RAM
    for x do
        PicOut = Foreground * Alpha + Background*(1-Alpha)
        Repeat two more times
        if State then
            DoSomething
            return value
```

Reducing this step by step, we end up with the For-loop as the dominant term, which equals to O(n) in complexity. As the other functions also has a For-loop as the highest order, they also equal to O(n), meaning that the complexity of the scene rises linearity with the number of pixels in horizontal direction.

To find a relative complexity for a scene, assume a set target for a pixel passthrough, meaning loading a clean pixel-line from RAM and into the line-buffer. By using this idea, it is possible to adjust the set target to get a line-budget that shows if it is possible to run certain operations within a given time window, given by the ratio between the pixel passthrough, and the different operations shown in Fig. 15 and Fig. 16. It show that, when compared to a simple passthrough, Alpha has a weight coefficient of 3.53 compared to passthrough, Mask has a 1.5 coefficient, and CLUT comes in at 1.21 times a pass-operation, on a per-pixel basis. By allocating a set value as a target for the line, it is possible to use these values to calculate the possible weight of the scene for any given composition.

As an example, lets assume a screen at a set number of horizontal pixels displays a simple picture at 60 frames per second at a set CPU frequency, but it is decided that it should rather display at 30 frames per second at the same frequency to allow the use of picture effects. The pixel passthrough had a cost of 50 at 60 frames per second, but as it is reduced to 30 frames per second, each line now has double the budget. By now using this new budget at 100, we can start theorizing how many operations we can afford to run at x pixels per line, given the weight ratios of the different operations.

Assuming the previous example, with 100 in budget, and a 50 pixel width, an example scene could be composed of 15 pixels applied with an alpha, 15 with a mask, and 20 with a CLUT. This leaves a budget of 0,35 left, which should be able to make the timings compared to the previous passthrough solution. A simple formula for calculating if the given combination is within its budget can be given by

$$Budget >= n * AlphaWeight + m * MaskWeight + o * CLUTWeight \tag{4}$$

by making sure that the equation is either balanced, or in favour of the budget.

Considering this, there are some limitations in the current implementation compared to a more realistic situation, as the software is supposed to be ran on a Micro-controller Unit. As the code is ran on a computer, it greatly reduces the run-time when compared to the more realistic speeds of the slower micro-controller. The implementation is also compiler-level dependent, meaning that which compiler is used has an effect on the outcome, as the model is high-level. Different compilers could give different results, closing in or spreading more out from the reference model. To decrease the effects of this, it is highly advisable to implement any further development of the model at a lower level language, such as C or Assembly, such that the compiler could potentially have a smaller effect on the performance, given that it does have an effect. This also follows more optimizations to the line buffer code, as the higher degree of control with low-level languages allows for clever manipulative tricks on the data sets, increasing performance.

The suggested implementation in this thesis, however, is meant as a research of viability, and has shown results that gives a vague idea if it is something to look more into, especially at the aforementioned lower level implementation.

6 Conclusion

This thesis has covered a suggested implementation of a high-level model for implementing a line-based rendering technique, using a scene descriptor to build the graphical interface. A description of the algorithm used in the implementation has been described at a higher level, and multiple tests have been performed to analyze the performance. A reference model, based on more traditional render techniques was also implemented, for comparison.

The model improved the RAM usage drastically by using only one line at a time, but seems to currently be limited by execution time when compared to the reference model. This could be due to several reasons, such as not implementing the model at a lower level such as C or Assembly, poor optimization in the implemented code, or compiler variations between the tested models.

From the model, a more general budget method was suggested for analyzing run-time conditions, allowing the designer to calculate if the scene can be rendered in time, given the complexity of the scene and the size of the screen.

References

[1] André R. S. Marçal, "Automatic color indexing of hierarchically structured classified images" [Online]. Available: https://www.researchgate.net/publication/4183583_Automatic_color_indexing_of_hierarchically _structured_classified_images

[2] Matthias Trapp, Sebastian Pasewaldt, Jürgen Döllner, "Techniques for GPU-based Color Quantization" [Online]. Available: https://www.researchgate.net/publication/332912124_Techniques_for_GPU-based_Color_Quantization

[3] Nvidia, "Chapter 24. Using Lookup Tables to Accelerate Color Transformations" [Online]. Available: https://developer.nvidia.com/gpugems/gpugems2/part-iii-high-quality-rendering/chapter-24-using-lookup-tables-accelerate-color

[4] StackOverflow, "HTML5 Canvas Creative Alpha-Blending" [Online]. Available https://stackoverflow.com/questions/17418048/html5-canvas-creative-alpha-blending [Accessed 03.06.2022]

[5] Asoke Nath, Sudipta Maji, "Scope and Issues in Alpha Compositing Technology" [Online]. Available: https://www.researchgate.net/profile/Asoke-Nath-4/publication/288838744_Scope_and_Issues_in_Alpha_Compositing_Technology/links /5686a63208ae197583975758/Scope-and-Issues-in-Alpha-Compositing-Technology.pdf

[6] Farid Pazhoohi, Alan Kingstone, "The Effect of Movie Frame Rate on Viewer Preference: An Eye Tracking Study" [Online]. Available: https://www.researchgate.net/publication/348679933_The_Effect_of_Movie_Frame_Rate_on _Viewer_Preference_An_Eye_Tracking_Study

[7] Kelvin Salton do Prado, "Understanding time complexity with Python examples" [Online]. Available: https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

[8] Ákos Szabó, "Panoramic Photography of Green Field" [Online]. Available: https://www.pexels.com/photo/panoramic-photography-of-green-field-440731/

[9] Christian Heitz, "Brown and Green Mountain View Photo" [Online]. Available: https://www.pexels.com/photo/brown-and-green-mountain-view-photo-842711/

[10] Irina Iriser, "Closeup Photography of Brown Wheat Field" [Online]. Available: https://www.pexels.com/photo/closeup-photography-of-brown-wheat-field-1301537/

A Results and pictures

A.1 Alpha, Mask and CLUT operators per line

A.1.1 Alpha

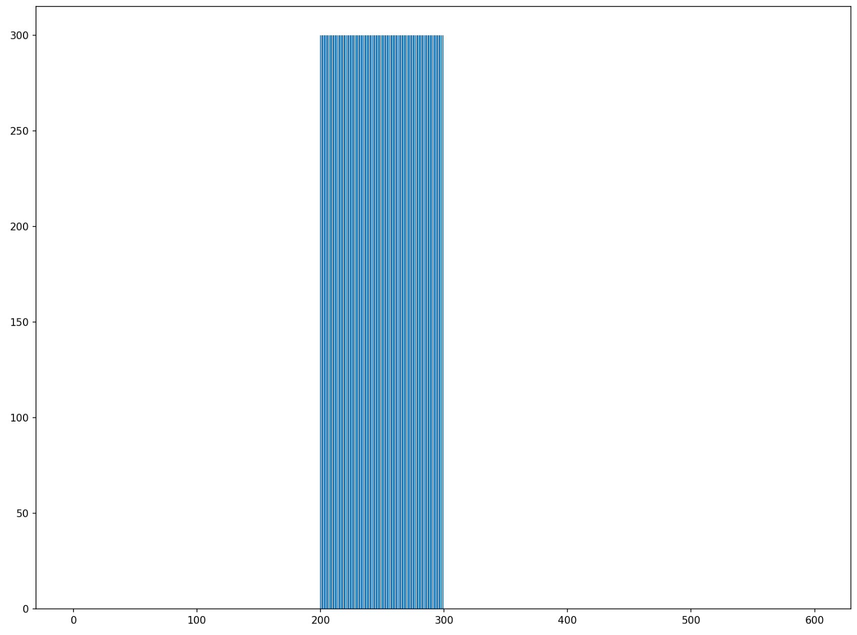


Figure 17: Scene 1 Alpha Usage

A.1.2 Masking

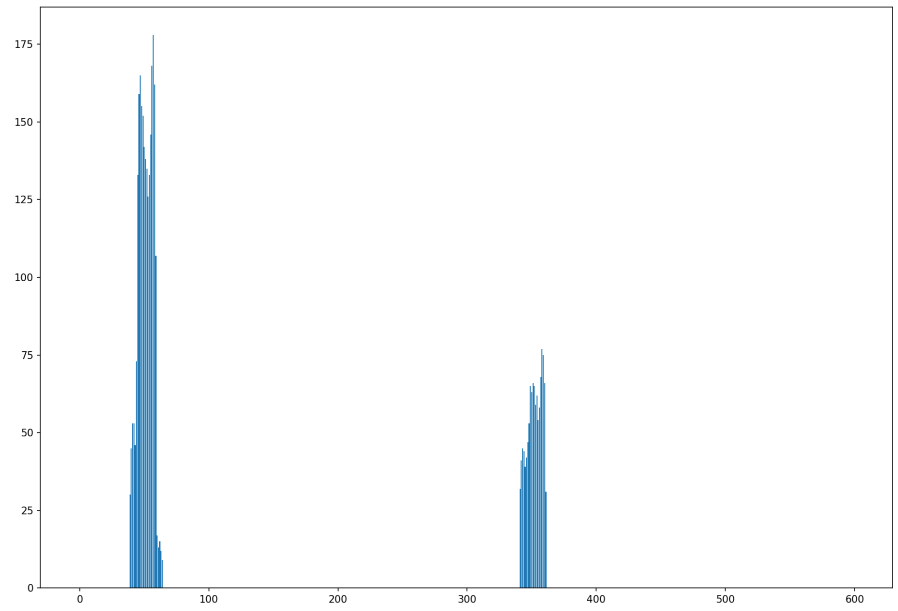


Figure 18: Scene 1 Mask Usage

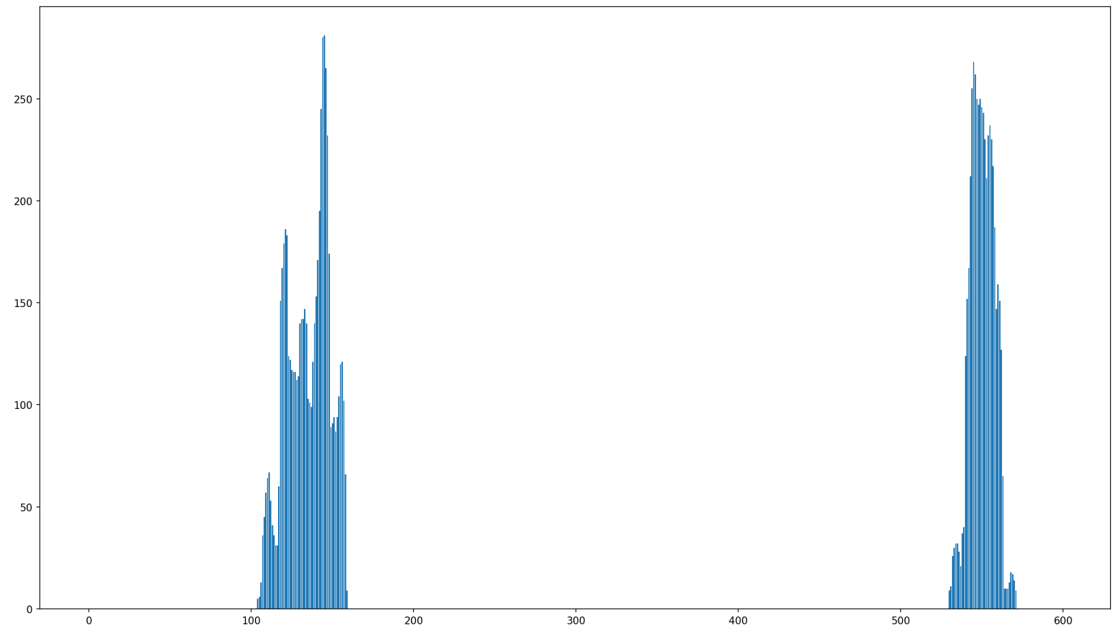


Figure 19: Scene 2 Mask Usage

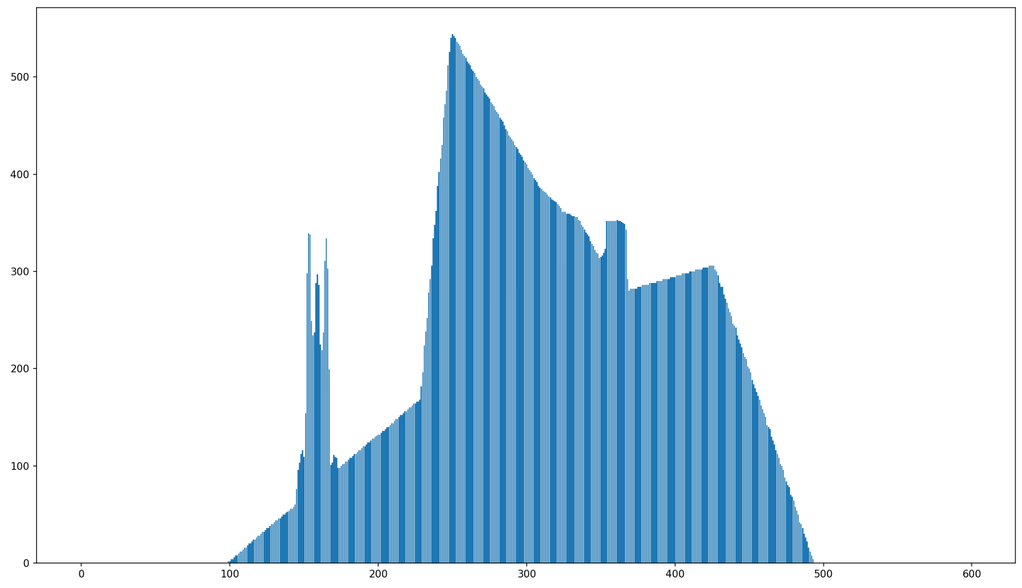


Figure 20: Scene 3 Mask Usage

A.1.3 CLUT

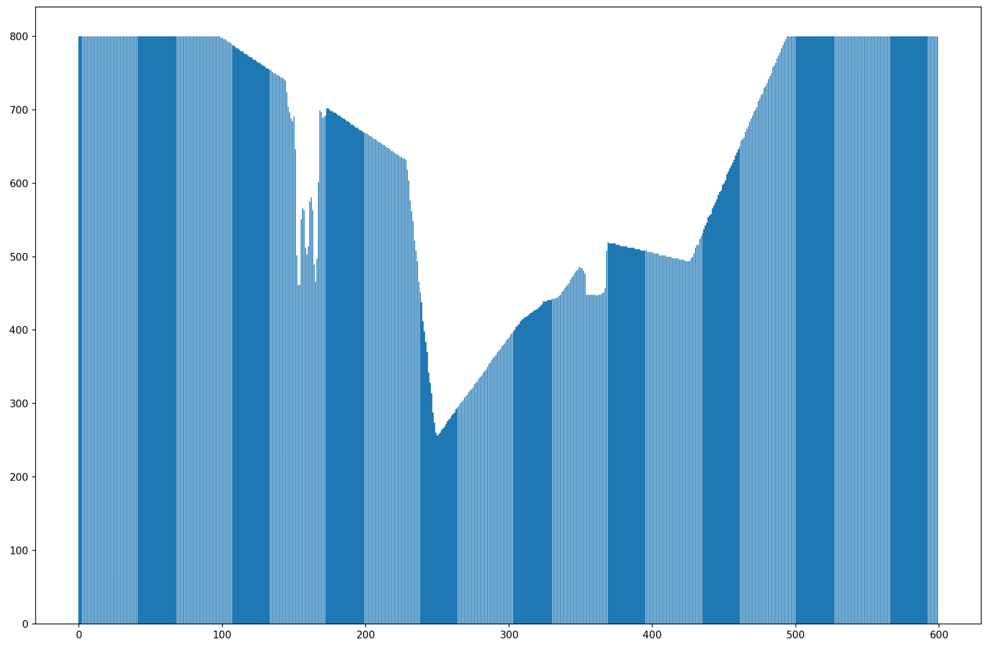


Figure 21: Scene 3 CLUT Usage

A.2 RAM put and get

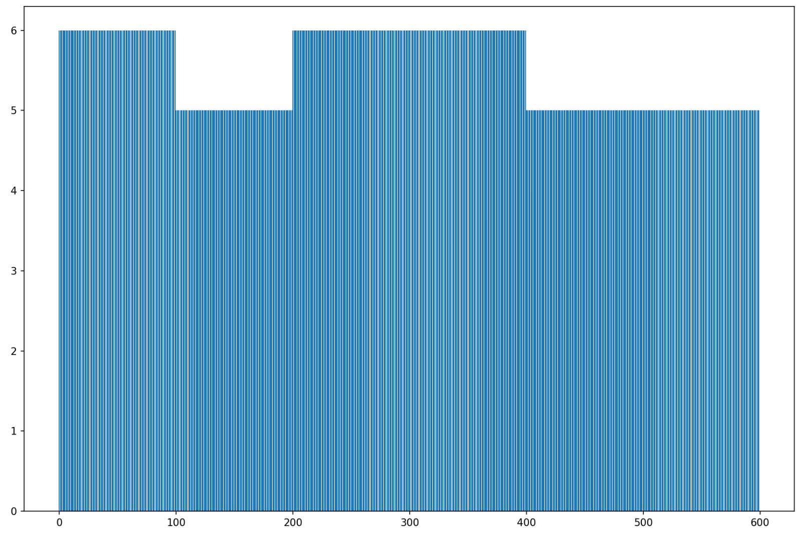


Figure 22: Scene 1 RAM puts

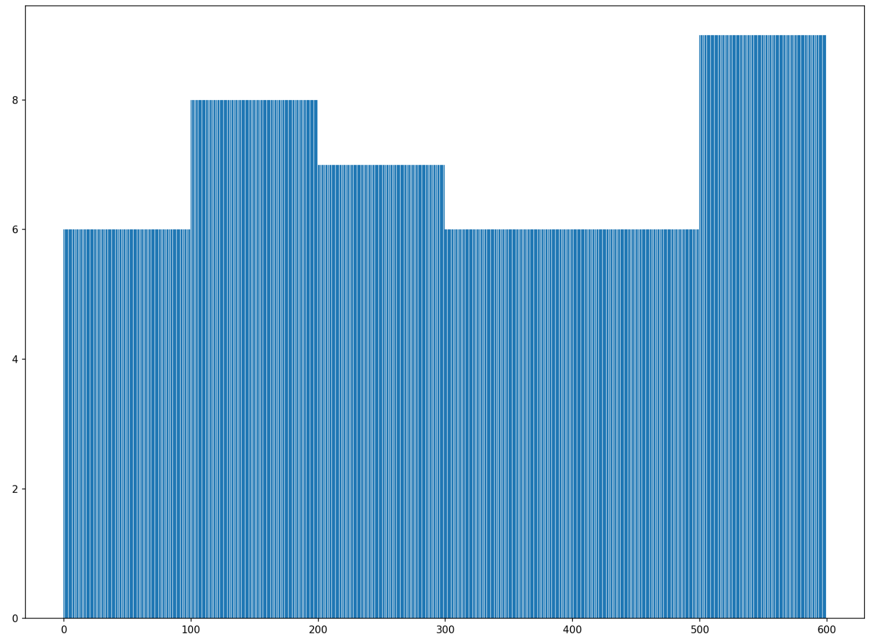


Figure 23: Scene 2 RAM puts

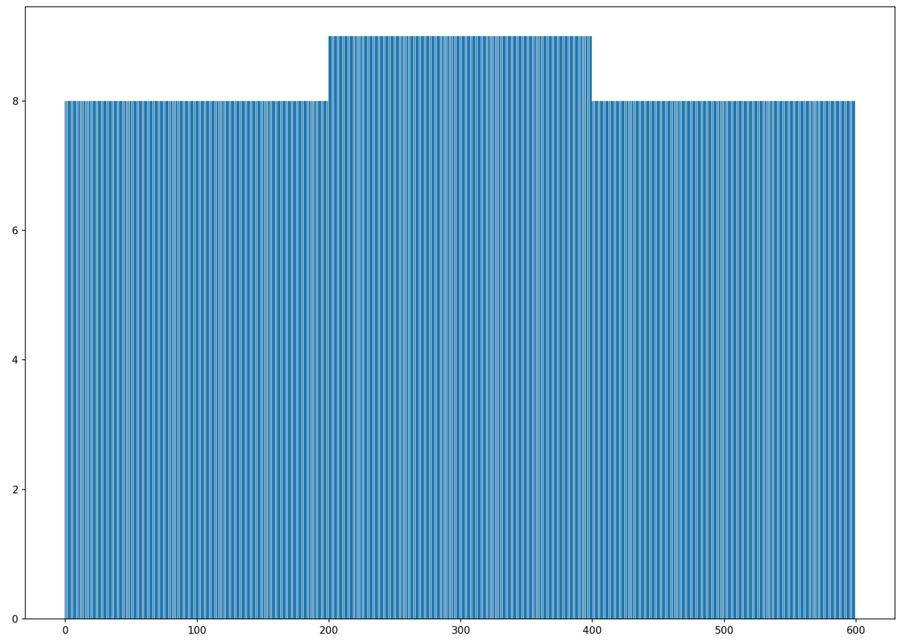


Figure 24: Scene 3 RAM puts

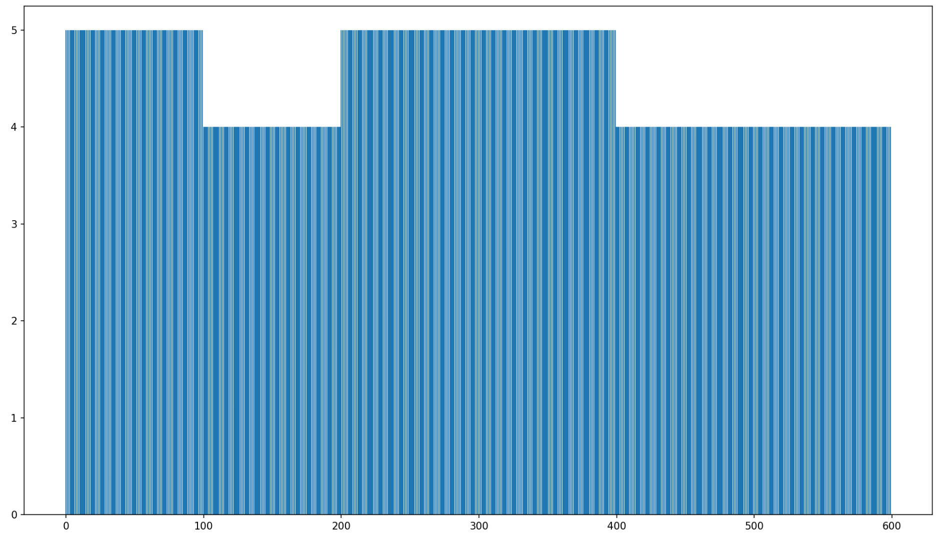


Figure 25: Scene 1 RAM gets

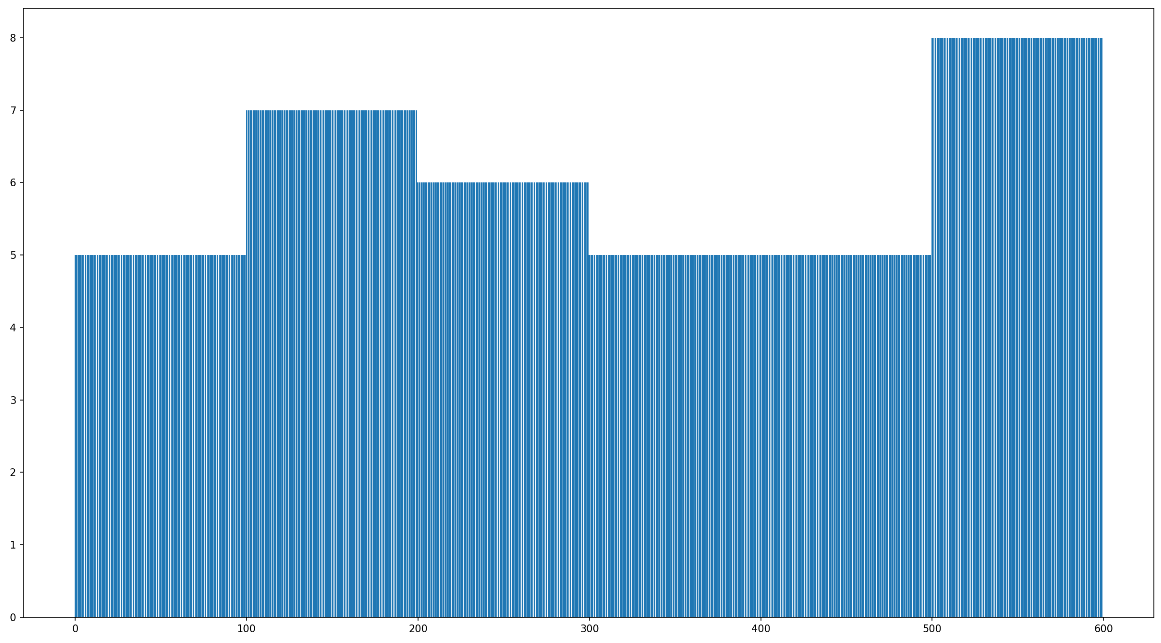


Figure 26: Scene 2 RAM gets

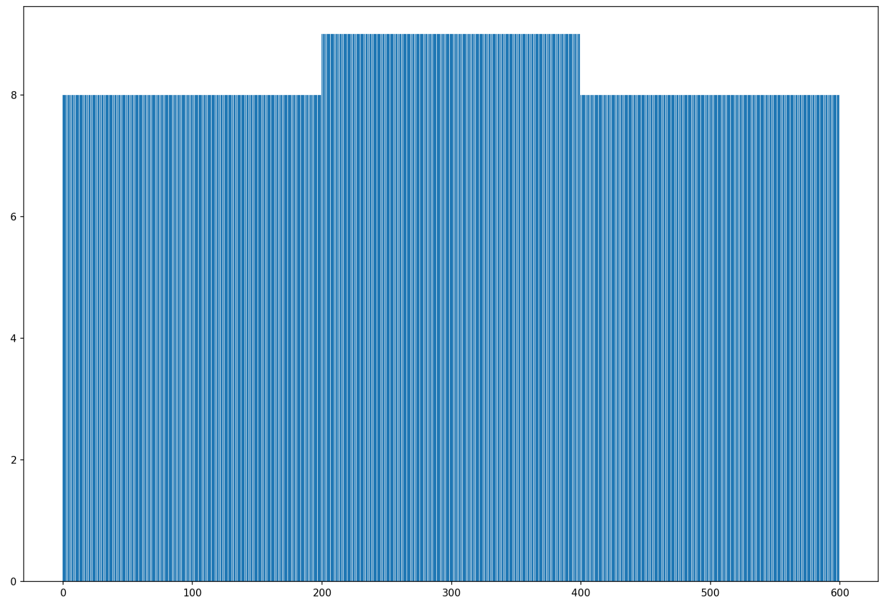


Figure 27: Scene 3 RAM gets

A.3 Output scenes

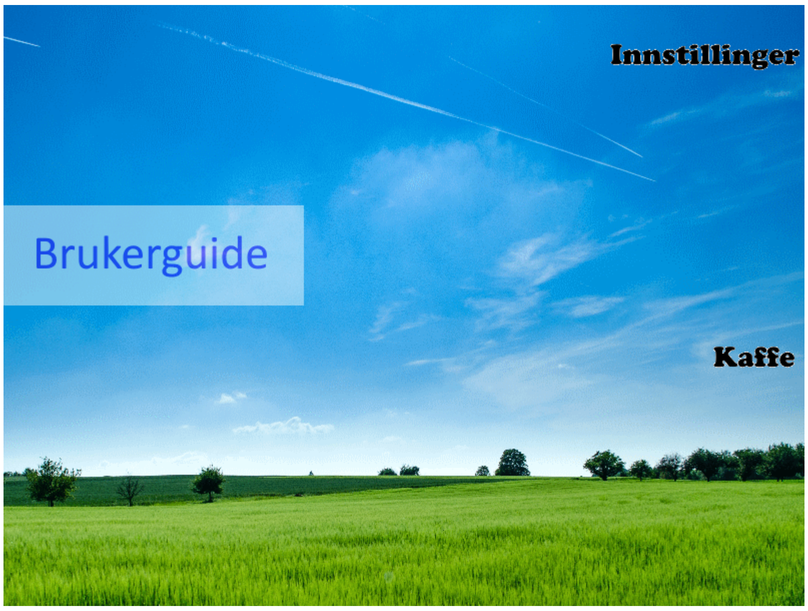


Figure 28: Scene 1 after construction

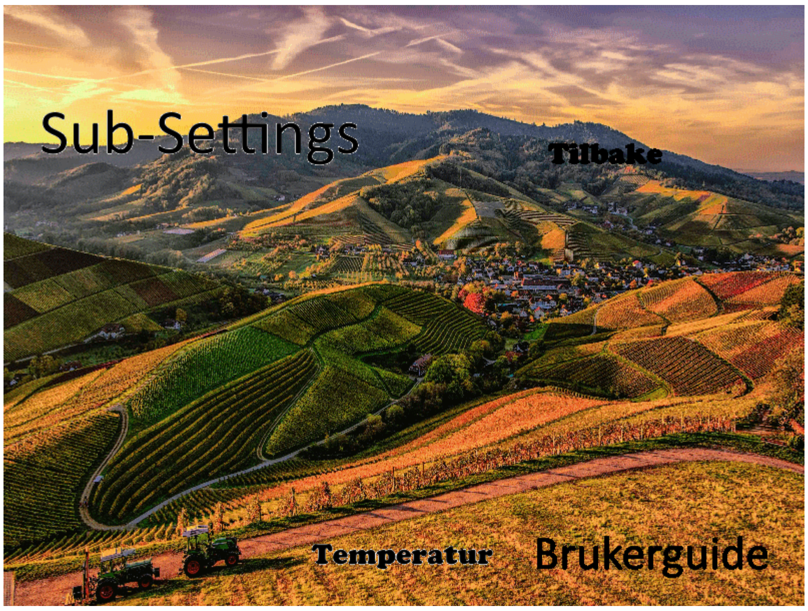


Figure 29: Scene 2 after construction

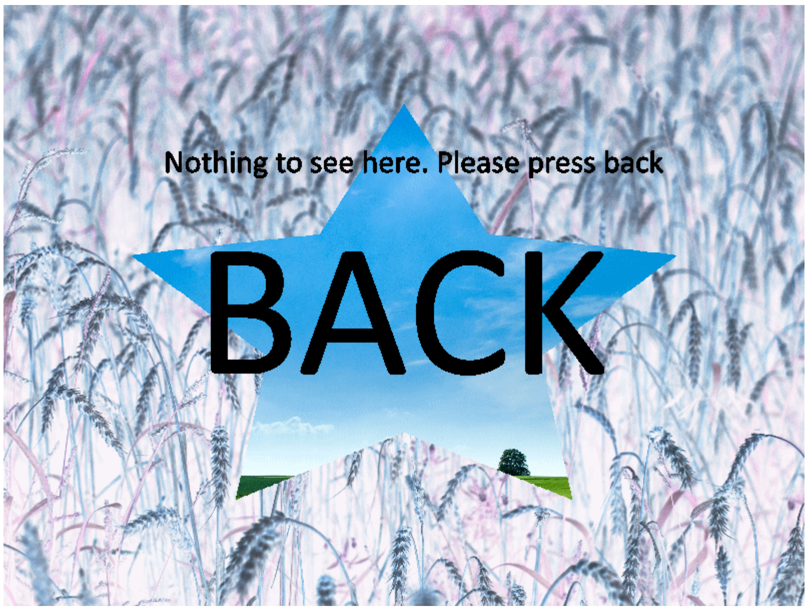


Figure 30: Scene 3 after construction

A.4 Scene components

This section will not include the text-image-files, as it does not seem necessary. They simply consist of black text on a white background. The mask is the same text-picture used again.

A.4.1 Backgrounds

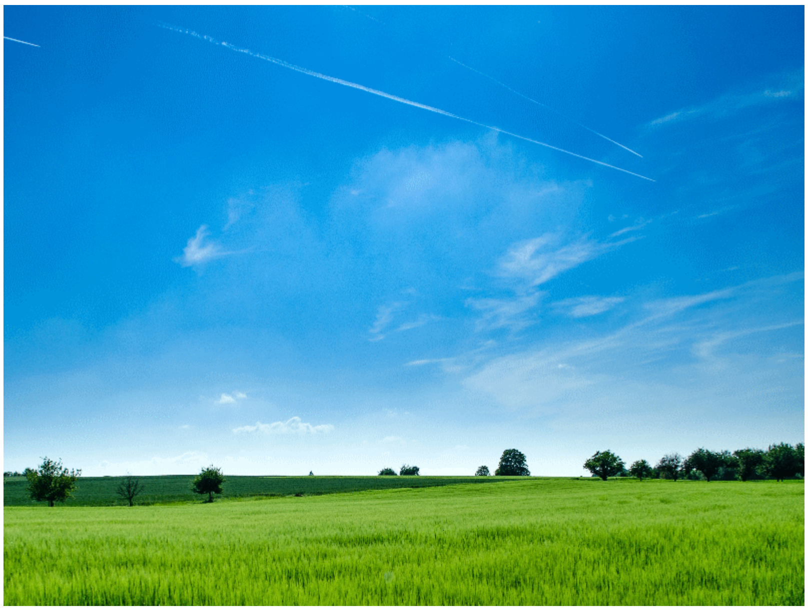


Figure 31: Background for scene 1. Collected from [8]

Figure 32: Background for scene 2. Collected from [9]

Figure 33: Background for scene 3. Collected from [10]

A.4.2 Masks

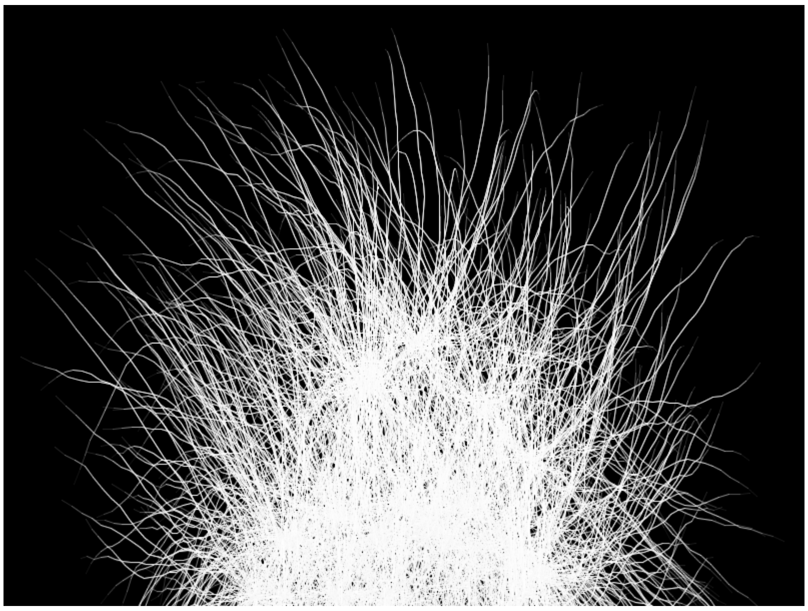


Figure 34: Grass mask

Figure 35: Star mask

A.4.3 Other testing results

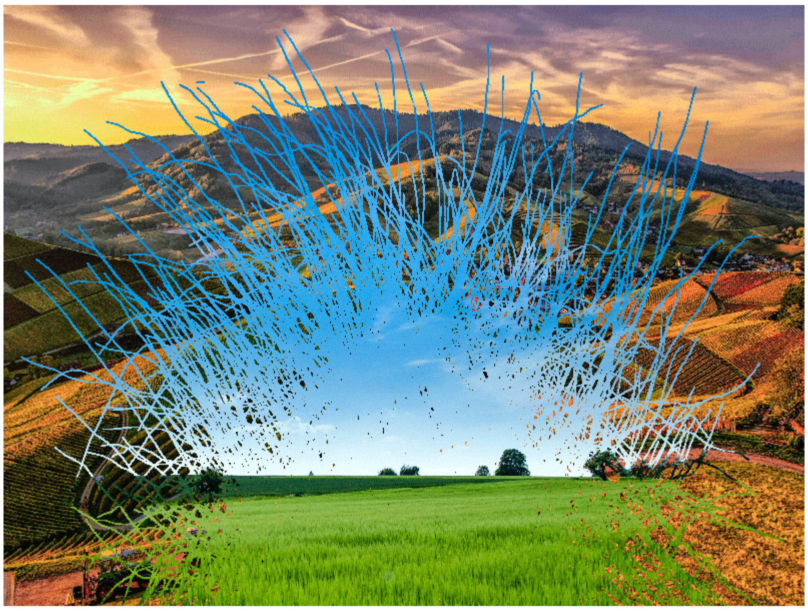


Figure 36: Masking

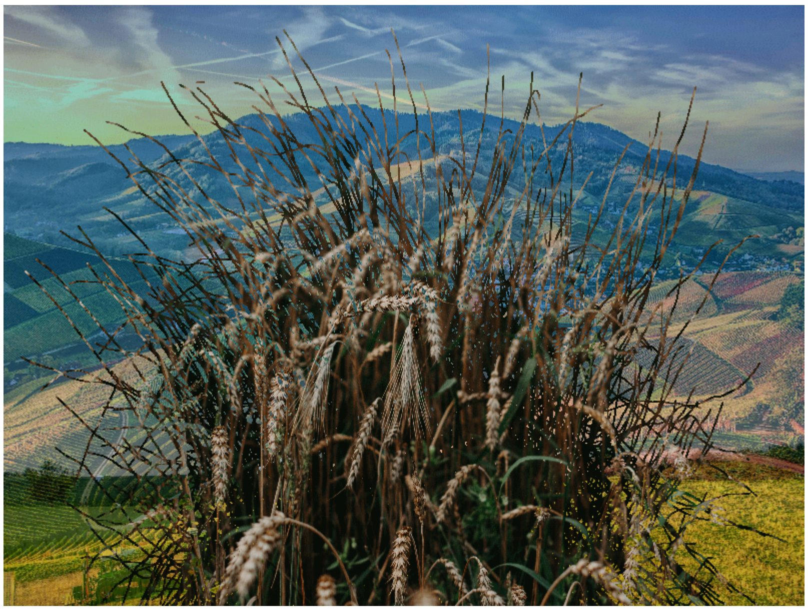


Figure 37: Masking and Alpha

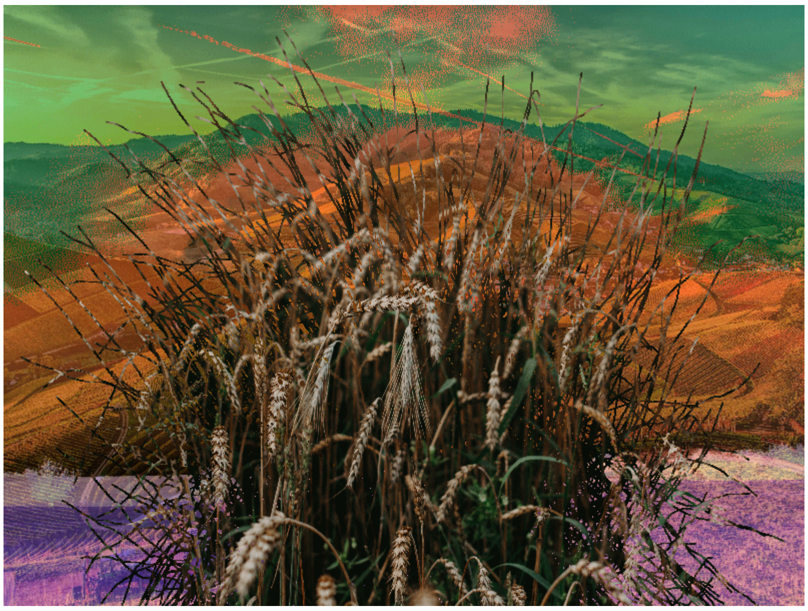


Figure 38: Masking and Alpha and CLUT

B Pre-project report for Master Thesis

Memory-efficient GUI display driving

Technical report
Runar André Saure
Trondheim, 2021

CANDIDATES ():				
Saure, Runar				
DATE:	**SUBJECT:**	**GROUP (name/nr):**	**PAGES/APDX:**	**BIBL. NR:**
19.12.21	TFE4590		18 /	N/A
subject teacher(s):				
Snorre Aunet, Glenn Ruben Bakke				
TITLE:				
Memory-efficient GUI display driving				

Summary:

This report aims to show the implementation of a memory optimized GPU driver, through usage of a scene descriptor and a line buffer to represent the graphics in a much more memory-conscious way than what standard models use. The resulting analysis show that improvements are made, and further optimization of code combined with restructuring and newer solutions can give an increase in the memory efficiency of such a system.

Content

1 Introduction - Description of Task

1.1 Memory-efficient GUI display driving

The task sets out to explore the possibilities of increasing the efficiency of memory usage in GUI-driving MCUs, through different solutions and code optimizations, where this paper represents one of the possible implementations. The implementation must be compared to a reference model, which consists of a normal implementation of a double-buffered as to get an idea of how viable the solution is compared to the reference.

The task is split into multiple smaller parts, which gives an idea of what steps must be taken to build the code. A Scene descriptor format is required to describe how the layering of any pictures is handled, and how they are passed on to the graphical processing part of the model. RAM- and CLUT operations are also required, as a proof-of-concept of the implementation.

To check the validity of the implementation, it is required to run the code with an error-free result, and be able to collect or draw the complete scene with all layers represented, with results that can be analysed for both the reference model and the suggested implementation.

The analysis requires a representation of how the memory usage differs between multiple scenes of different complexity, especially compared to the reference model.

2 Background - Theory

2.1 Colour Lookup-Table

A simple way to add some modification to a picture is by using a Colour Lookup-Table (Hereby referenced to as CLUT) to modify the color characteristics of the picture.

CLUTs can be constructed either by using multiple 1D-arrays for each aspect of the picture such as the RGB values, or other picture modifications such as colour alpha. Another solution is to arrange it into a 3D cube, representing each R, G and B channel, which can easiest be visualized by a 2D-representation, as seen in Figure. 1 below.

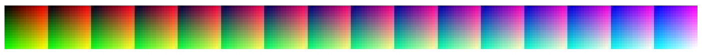


Figure 1: 3D-LUT can be represented as a 2D-array. Found at UnrealEngines Website[1]

By locking one of the colour components of RGB as the Z-axis, the two other are shown on the X and Y axis. by iterating over the Z-axis, here shown as the array of pictures with a horizontal offset, we get a better understanding of how increasing the Z-component affects the output picture when calibrating with the X and Y-component. together, the RGB values will extract one single "dot" from the 3D-LUT, giving the coloration of the pixel on the display.

A 1D implementation is much simpler, with each channel represented by a separate indiscriminate array. Each input is specifically mapped to a value in the array, while still keeping a high precision in colour accuracy when applying the CLUT. Another advantage of a 1D LUT is the small size required, where the amounts of bits needed is given by

$$LUT_{Size} = 2^{numberofbits} * N_{numberofchannels} \tag{1}$$

Any remapping or adjusting of the 1D LUT also requires much simpler functions compared to a 3D-LUT.

The main purpose of LUT however, is to modify a picture to give a hue that differs from the original. For example, by limiting the blue and green component from a custom LUT, will when applied result in a much more red tinted picture on the screen, or a yellow tint as seen in . By changing the LUT, either by importing a new one or doing mathematical manipulations on the array, it is possible to create many different filters, depending on the desired colour alteration or effects.

2.2 Scene Description

A scene descriptor is a data structure that lays out the representation of how graphics can be displayed. Its structure is inherently hierarchical, where each "layer" will describe a component of the graphics to be rendered. This is then used to communicate with external processes or functions in what manner any graphical operations or manipulations will be handled.

By using a scene descriptor, it is possible do to on-the-fly manipulations of which layer to draw, and more easily modify attributes such as xy-placement, CLUTs and layer of the scene, without disrupting the whole frame.

2.3 RAM

The RAM module is one of the main focus points of the task, as the project sets out to minimize its usage. RAM, short for Random Access Memory, is an intermediate storage unit in MCUs, as a mid-level between cache found inside the CPUs, and any other storage such as flash, SSD or EEPROM. The RAM remains volatile however, and any content will disappear during a power outage, so correctly storing any valuable variable is highly recommended.

To represent RAM in an easy way, it is best to imagine it as a long array, with each cell containing some amount of data, usually up to 8 bits. These are known as registers, with each having an unique address. These addresses are referenced to when writing to or reading the target register. Usually, larger data sets are spread over multiple registers, and will be preferred to be spatially placed, as to reduce the complexity of placing and retrieving.

2.4 Frame Buffering

A Frame buffer is a dedicated portion of the RAM that is specialized in containing graphical I/O data for graphical processing of the system. The data stored inside the buffer is used to drive a screen with information or GUI, Which can be represented as graphics, text or similar models. In modern systems, the frame buffer contains at least one picture ready to be rendered, represented by the pixel values for RGB, Alpha etc., usually in a bitmap format. As soon as a picture, or "frame" is ready and the GPU is ready for the next frame, the frame buffer sends the current frame to the graphical processor, and starts retrieving the next frame.

2.5 Double Buffering

In modern systems, double buffering is a method often implemented to reduce the amount of flickering experienced when using any screen to interact with the MCU. If the next frame is not ready before the GPU requests the next one, commonly caused by incorrect timings or too complex frame operations, the screen gives the illusion of "blinking" as it tries to keep up with the drawing demand of the system. By using a double buffer, the processor completes a picture inside a dedicated memory space known as a buffer instead of directly to the screen, which gives the processor time to finish the picture before the graphics processor starts rendering the frame. This way, the display will not stutter as it awaits the data from a slower CPU, but can keep up to the faster GPU and give a smoother user experience.

By using the double buffering method however, the amount of memory required for driving the screen flicker-free increases, and can become quite "big" in cases where larger or more detailed pictures are used. This poses a big problem in MCUs where memory space is very limited, and using every bit of space wisely is advisable.

2.6 Line Buffer

The Line Buffer is a solution that attempts to mitigate the problem of space usage introduced by buffering. By limiting the buffer size to only one line, or "width" of the screen at a time, it is possible to greatly reduce the size of the buffer, while still avoiding problems associated with a bufferless rendering, such as flickering.

2.7 Python Packages

Following are some key packages used in the code. It is highly advised to read up on these if trying to replicate the results achieved in this report. They can be found at Numpy[4], PIL[3] and tkinter[5]

2.7.1 Numpy

Numpy is a very commonly used library of Python, and will therefore be kept very short, but is included as it has been important for achieving a solution. Numpy is a package that contains multiple mathematical functions and operations, and has many useful functions for operating on array objects, especially arrays with multiple dimensions, which stands for a lot of the operations done in the solution. The advantage of Numpy is also its speed. Compared to standard manual solutions, it is optimized to achieve the fastest possible output.

2.7.2 PIL

PIL, "Python Image Library", or "Pillow" as the newest fork is called, is a library specialized in handling images in code, with a wide support of formats and processing operations. It is optimized for fast data access, and allows a wide amount of picture-related operations for handling any picture to be used in the scene descriptor. Its main use for the task is focused on importing and exporting the pictures, and transforming the format, in this case JPGs, into modifiable arrays.

2.7.3 tkinter

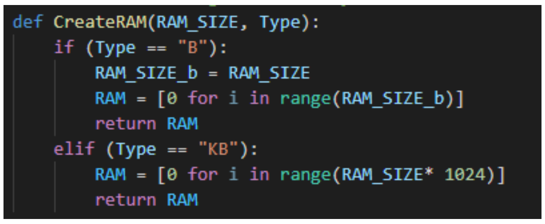
tkinter is a lightweight and easy-to-use GUI package for python, which allows displaying of graphics without the use of external MCUs. This package has mainly been used in the task for displaying and checking the correctness of the data traveling through the system. Without a GUI package such as tkinter, testing and analyzing becomes much harder as some appliances, such as CLUT, requires more work to verify if one checks the data directly instead.

3 Methodology

This section will explain some of the implementations of the solution model. Simple code snippets will appear for simple functions, while the more complex solutions will be explained in a more abstract manner.

3.1 RAM Implementation

The RAM object was chosen to be represented as an array to simulate the closeness to its physical representation with registers and addresses. A simple instantiating is shown by Fig. ?? , where the RAM is sized by an input variable RAM_Size, which allows to easily resize or create multiple RAM instances on the fly, and "Type" allows you to input the value in both Bytes and Kilobytes.



```
def CreateRAM(RAM_SIZE, Type):
    if (Type == "B"):
        RAM_SIZE_b = RAM_SIZE
        RAM = [0 for i in range(RAM_SIZE_b)]
        return RAM
    elif (Type == "KB"):
        RAM = [0 for i in range(RAM_SIZE* 1024)]
        return RAM
```

To allow sector to exist inside the RAM, a simple function ram_allocate_sectors(Layers, RAMSize) was created, which allows the user to split the size of the RAM into multiple zones, each representing the memory region for a single layer. A return value "SectionSize" is used as an offset for any accesses to the memory.

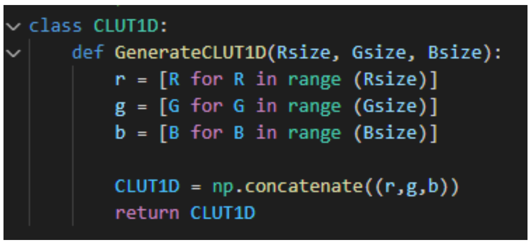
Some important functions for the RAM are the put and get functions, which each has a function instance for the data and the CLUT. A ram_put_data(Layer, Offset, Data, RAMIn) function allows the user to designate which layer the chosen data is saved in, and in which RAM-instance. This function manipulates the picture array to be saved inside the RAM in a spatial way, allowing quick access during a get.

The corresponding ram_put_clut acts in a similar manner, but uses a return value DataLen from the data function to determine the additional offset of the CLUT, such that the CLUT begins where the Data ends.

The inverse functions ram_get_data and ram_get_clut works in the opposite way of the puts, retrieving the data arrays in a similar reversed manner, and returns the collected values.

3.2 CLUT Implementation

The CLUT used in this task is a 1-Dimensional LUT, with a simple basis generate code for a linearly scaled array for each RGB-value, allowing for an easy-to-use easy-to-modify LUT, which still retains the accuracy while having a very small footprint.



```
class CLUT1D:
    def GenerateCLUT1D(Rsize, Gsize, Bsize):
        r = [R for R in range (Rsize)]
        g = [G for G in range (Gsize)]
        b = [B for B in range (Bsize)]

        CLUT1D = np.concatenate((r,g,b))
        return CLUT1D
```

To apply the CLUT, a simple function ApplyCLUT(Data, CLUT) allows the user to change the color saturation of the picture. The function checks every R, G and B value from the input data, and compares it to the input CLUT. By finding the corresponding R,G or B value in the CLUT, it fetches corresponding modified new value from the CLUT, and applies it to the data.

An attempt to make a 3D-LUT was made during the process, but was put aside to prioritize a simple working LUT, however it should be possible to re-implement the 3D-LUT with some more work, which would allow for much more customizability for the LUT. Any future iterations might switch to the 3D-LUT, but considerations must be taken to as whether the advantages of a 3D-LUT is worth the drawback of a much larger memory footprint.

3.3 Scene Descriptor

The Scene Descriptor is a class which contains the values of every layer, in addition attaching the CLUT and functions to checking if CLUT is to be applied. The Scene file can be checked for any layer numbers, and the placement of the pictures in each layer.

When printing the scene to a display, the scene first collects the amount of layers involved in the following line. If multiple layers are found, it saves the values for the width of the layers, and checks the X-position of the topmost layers. This tells the scene descriptor where to start and stop drawing each layer. The process starts with the lowest layer number, with layer 0 being regarded as the background for the scene. As the Descriptor reaches the starting parameters for any higher layer, it switches the picture to be drawn to the topmost of the ones on the current co-ordinate, and renders it until the end. When the end of line is reached, the resulting line is sent to the linebuffer, and it continues by iterating the y-axis onto the next line.

3.4 Linebuffer and Buffer

The linebuffer is generated form a simple array-generation, and then uses a function FillLineBufferFromArray to apply the input data, after any operations such as layering, CLUT etc. is done, to the 2-D array.

The regular buffer is replicated in the same manner, but with a third dimension added to simulate a full screen, which considerably increases the size. To replicate the buffer in a simple manner, the FillLineBufferFromArray function can be used to send into the buffer, iterating over the y-size of the regular buffer array.

4 Results

This section will present some of the measurements and analyses extracted from the code, with a short explanatory text.

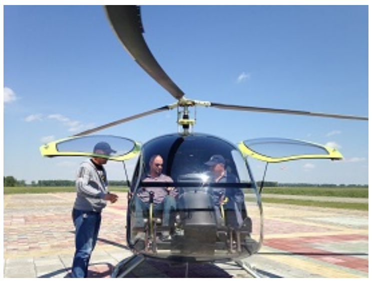


Figure 2: Reference picture

Figure 2 show a reference picture collected from WikiMedia[6].

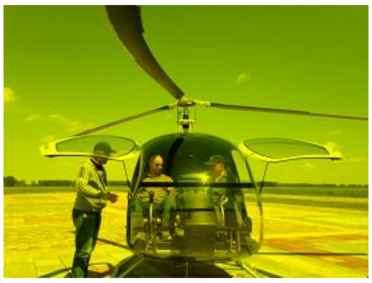


Figure 3: CLUT Applied

Figure 3 show the same picture after a yellow-based CLUT tint is applied.

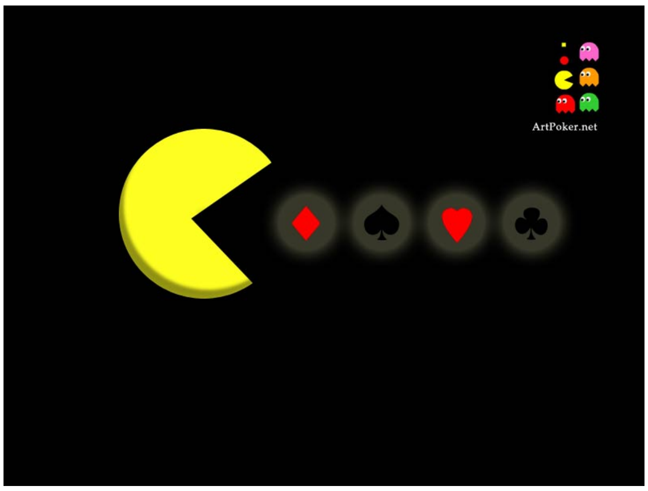


Figure 4: Larger reference picture

For further testing of the system, a larger picture shown in fig. 4 is used to verify that the model can handle larger pictures.

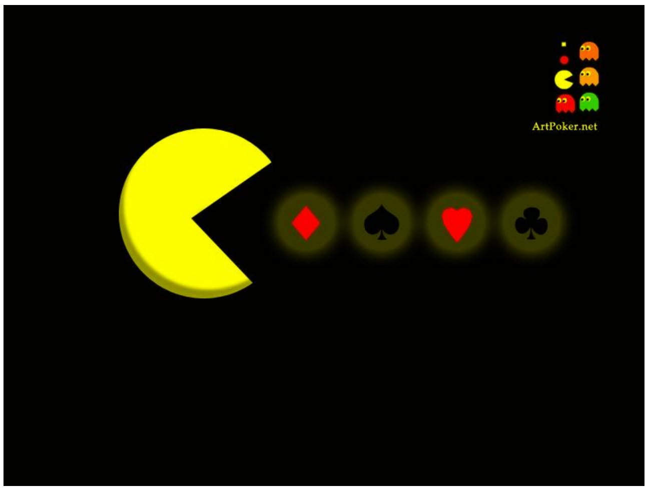


Figure 5: Larger reference with CLUT

Fig. 5 uses the same yellow-tint CLUT as shown in fig. 3

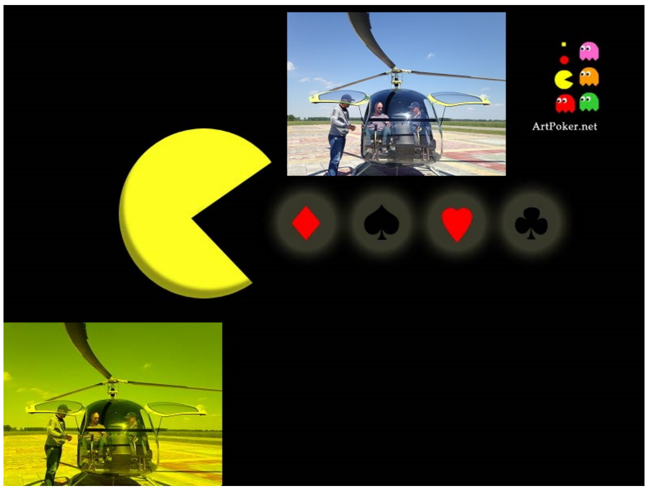


Figure 6: Complex scene build-up

To test the capability of the scene descriptor, fig. 6 show a constructed scene showing a background (Pacman), an offset picture (Upper helicopter), and an offset CLUT-applied picture (Lower helicopter).

Fig. 7 shows the amount of RAM used when comparing a Linebuffer solution to a Double buffer solution when comparing a 800x600 @24bpp picture.

Fig. 8 show the comparison of a 273x204 @24bpp picture

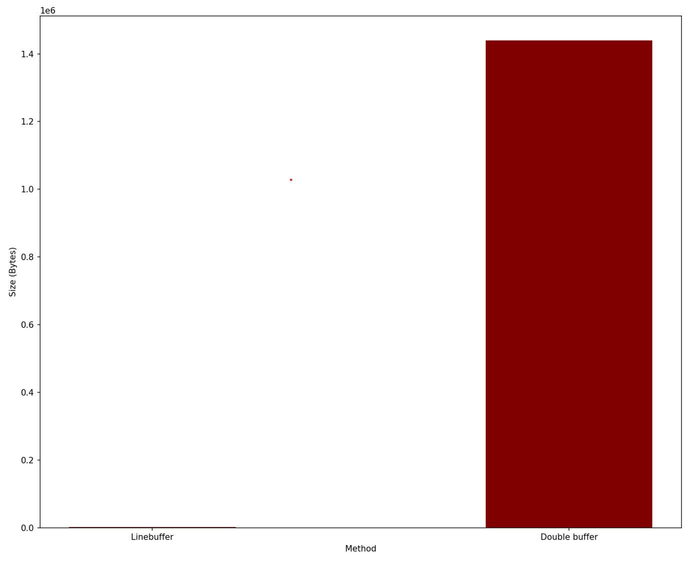


Figure 7: Memory usage of a 800x600 24bpp picture

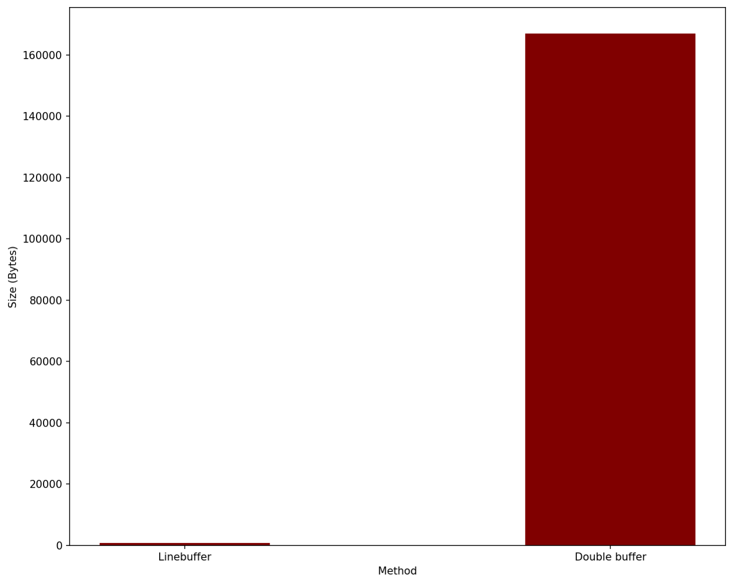


Figure 8: Memory usage of a 273x204 24bpp picture

5 Discussion

Throughout the results, the validity of the model is represented throughout the pictures. Both the process of saving the picture successfully to ram, retrieving it and applying the CLUT filter shows basic functionality as could be required for a solution. The more interesting case comes with Fig. 6, where the Scene Descriptor comes into play.

By utilizing the descriptor, the code allows for the X-Y offset of pictures, where it is freely possible to place the pictures within the bounds of the predetermined screen size. The layering is also shown, where the scene descriptor explained in Chapter 3.3 layers the two helicopter pictures on top of the background picture. By introducing alpha masking, it would be possible to allow for more advanced figures, such as the Spadille and other figures seen on the Pacman-example.

When comparing the results of Fig 7 and Fig. 8, it shows that the line buffer gives a lower memory footprint compared to the double buffer solution when using larger pictures. This might however be affected by the aspect ratio of the pictures, and the gain of linebuffer when comparing the first and second results, it does not increase linearly, as the gain is much lower.

As can be seen from Fig. 7, the linebuffer consumes a considerable less amount of RAM when compared to the double buffer. Further optimizations could have made the double buffer size considerably smaller, but it would still not reach the low footprint of a linebuffer solution. Regarding this, the most pressing issue is the analysis of the bandwidth of such a solution, but that part remains as a further development note. A size reduction does not help if it struggles to send the data in time, which will quickly result in a sluggish experience for the user of the display. The least severe symptoms would be "flickering" or being able to see the lines being drawn, but it could degrade into clearly visible drawing of each line. Further testing and work will show if the solution is applicable for real-world hardware.

6 Further development and the road ahead

This project represent a limited implementation of the solution, but with many ways to expand. One of the most considerable improvements are various compression algorithms to reduce the size of both pictures, CLUT and the buffer even further. the addition of an on-the-fly generated CLUT could also help save memory space, depending on if it has a 1D, 2D or 3D-implementation of the CLUT. The 3D-implementation will have the largest footprint reduction from this, as it is considerably larger than a 1D-array.

The scene descriptor has more potential for expansion, allowing for more complex picture manipulation such as alpha blending, masking, color space reduction and expansions, linear interpolating of the CLUT and more advanced techniques.

The code can also be expanded to be tested on an actual MCU, which would allow more MCU-specific optimizations, and make way for analyzing bandwidth, processor cycles, memory operations and similar, which will give a much wider view of the viability of the project on actual hardware.

7 Conclusion

The experiment of implementing a solution to reduce the memory footprint of GPU driving has shown itself to have potential. The solution has proved to be able to operate around a self-implemented RAM-module and appliance of CLUT to the picture, but also more advanced tasks such as layering and placement of pictures, with regards to the drawing of a single line at a time.

More work is needed to prove that this is a viable model for actual implementation in hardware, but with more advanced features, optimizations and testing, it could give results that shows the merit of the model.

References

[1] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/PostProcessEffects/UsingLUTs

[2] Further reading:

[3] https://pillow.readthedocs.io/en/stable/

[4] https://numpy.org/doc/stable/index.html

[5] https://wiki.python.org/moin/TkInter

[6] Helicopter Photo: https://commons.wikimedia.org/wiki/File:Small$_pict_test.JPG$

[7] Pacman Photo: http://www.artpoker.net/pacman-wallpaper-when-pacman-and-poker-come-together-1024x768

C Python Code for model and reference model

1 Code Files

1.1 AlphaBlending.py

```
#Alpha Compositing

import numpy as np

#Alpha Compositing is the process of combining two images with a transparency mask
PictureOut = np.zeros((800, 4), dtype=np.uint8)

#Function that checks if a picture contains alpha.
def check_alpha(picture, TestEntity):
        #If it does not, allow the user to input a custom alpha. If it is not within bou
        if (len(picture[0][0]) == 3):
            while True:
                alpha = input("Enter alpha value: ")
                if alpha.isdigit():
                    alpha = int(alpha)
                    if alpha >= 0 and alpha <= 255:
                        break
                    else:
                        print("Invalid alpha value")
                else:
                    print("Invalid alpha value")


            #put the alpha value into thepicture
            PictureOut = [[[0, 0, 0, 0] for i in range(len(picture[0]))] for j in range(

            for i in range(len(picture)):
                for j in range(len(picture[i])):
                    PictureOut[i][j][0] = picture[i][j][0]
                    PictureOut[i][j][1] = picture[i][j][1]
                    PictureOut[i][j][2] = picture[i][j][2]
                    PictureOut[i][j][3] = alpha
            print("Alpha value set.")
            return PictureOut
        else:
            print("Alpha already present")
            #ask user if want to input new alpha
            print("Do you want to input a new alpha value? Y/N")
            if (input() == "y" or "Y" or "yes" or "Yes"):
                while True:
                    alpha = input("Enter alpha value: ")
                    if alpha.isdigit():
                        alpha = int(alpha)
                        if alpha >= 0 and alpha <= 255:
                            break
                        else:
                            print("Invalid alpha value")
                    else:
                        print("Invalid alpha value")

                for i in range (len(picture)):
                    for j in range (len(picture[i])):
                        picture[i][j][3] = alpha
                    return picture



                else:
                    print("ok, no alpha value set.")

                return picture





#Formula for simple "Over" operator. replaced by simply plugging it directly into ApplyA
def AlphaFormula(Fg, Bg, Alpha):
        #Out = Fg*A+Bg*(1-A)
        return Fg*Alpha + Bg*(1-Alpha)

#Create a function that blends two pictures together, using foregrounds alpha as "mask".
def ApplyAlpha (Foreground, X_Offset, Operator, Background, TestEntity):
    #The input Foreground is a picture of x length with a x offset relative to backgroun
    #The output is a picture of the same length as the background, but with the foregrou
    if (Operator == ("Over" or "over")):
        PictureOut = np.zeros((len(Background), 4), dtype=np.uint8)
        #StartTime = time.time()
        #map the foreground alpha value to a range of 0 to 1 manually. This is done to a
        Alpha = Foreground[0][3]
        AlphaOut = Alpha
        Alpha = Alpha/255
        for CurrentX in range(len(Foreground)):


            #If the current pixel is free
            if TestEntity.FreeLine[CurrentX + X_Offset] == True:




                    TestEntity.ApplyAlpha_Pixel += 1
                    #Apply the alpha formula Fg*Alpha + Bg*(1-Alpha)
                    PictureOut[CurrentX+X_Offset][0] = Foreground[CurrentX][0]*Alpha + B
                    TestEntity.ApplyAlphaR += 1
                    PictureOut[CurrentX+X_Offset][1] = Foreground[CurrentX][1]*Alpha + B
                    TestEntity.ApplyAlphaG += 1
                    PictureOut[CurrentX+X_Offset][2] = Foreground[CurrentX][2]*Alpha + B
                    TestEntity.ApplyAlphaB += 1
                        #Combine Red, Green and Blue into PictureOut
                    PictureOut[CurrentX+X_Offset][3] = AlphaOut
                        #PictureOut[CurrentX] = [Red, Green, Blue, AlphaOut]
                    if TestEntity.CurrentLayer != 0:
                        TestEntity.FreeLine[CurrentX + X_Offset] = False

        TestEntity.AlphaBlend += 1
    else:
        return
    return PictureOut
```

```
#Put alpha value into the input picture, which is in format (800, 4). The input also con
def PutAlpha (Picture, Alpha):
    PictureOut = np.zeros((len(Picture),len(Picture[0]), 4), dtype=np.uint8)
    for j in range (len(Picture)):
        for i in range(len(Picture[0])):
            PictureOut[j][i][0] = Picture[j][i][0]
            PictureOut[j][i][1] = Picture[j][i][1]
            PictureOut[j][i][2] = Picture[j][i][2]
            PictureOut[j][i][3] = Alpha

    return PictureOut
```

1.2 AlphaMasking.py

```
#Masks each channel given by the mask.
import numpy as np





#Masks each channel given by the mask.
def ApplyMask(PictureFG, X_Offset, PictureBG, Mask, TestEntity):
    #White (>0) means draw, black (0) means remove
    #Create PictureOut with the same size as PictureFG.
    PictureOut = np.zeros((len(PictureBG), 4), dtype=np.uint8)


    #For each X
    for i in range(len(PictureFG)):
        if TestEntity.FreeLine[i+X_Offset] == True:
       #For each channel
                #If the mask is not transparent
                if Mask[i].all() == 0:
                    #Set the channel to the background picture
                    PictureOut[i+X_Offset] = PictureBG[i+X_Offset]
                    TestEntity.NoMask += 1
                    pass


                else:
                    #Set the channel to the foreground picture
                    PictureOut[i+X_Offset] = PictureFG[i]
                    if TestEntity.CurrentLayer != 0:
                        TestEntity.FreeLine[i+X_Offset] = False
                    TestEntity.ApplyMask += 1

    return PictureOut
    #, TestEntity.FreeLine





#Check if the picture is RGB or RGBA. Returns the number of channels. Not used.
def CheckNumbersOfChannels(Picture):
    if (len(Picture[0][0]) == 4):
        return 4
    elif (len(Picture[0][0]) == 3):
        return 3
    else:
        print(”No. Just no.”)
```

1.3 Analytics.py

```
import matplotlib.pyplot as plt




#Input is a class which contain multiple attributes which are arrays. Create a histogram
def histogram(input_structure, Length):
        #create a histogram for each attribute
        Histogram = [0]*Length
        for i in range(len(input_structure)):
            #create a histogram for each attribute
            Histogram[i] = input_structure[i]

        PlotHistogram(Histogram, Length)
        return Histogram


#Input a class. The class contains multiple atributes. Create a histogram for each attrib





#Function to plot the histogram
def PlotHistogram(Histogram, Length):
        plt.bar(range(Length), Histogram)
        plt.show()
        #Save the histogram
        #plt.savefig(”F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Billeder/Histogram

#Save every iteration-value into array
def Testing(TestEntity, CurrentY):


    TestEntity.ApplyAlpha_Pixel_Array[CurrentY] = TestEntity.ApplyAlpha_Pixel
    TestEntity.ApplyMask_Pixel_Array[CurrentY] = TestEntity.ApplyMask
    TestEntity.CLUT_Pixel_Array[CurrentY] = TestEntity.CLUT_Pixel_Applied
    TestEntity.RAM_put_Array[CurrentY] = TestEntity.RAM_put
    TestEntity.RAM_get_Array[CurrentY] = TestEntity.RAM_get
    TestEntity.RAM_Used_Bytes_Array[CurrentY] = TestEntity.RAM_Used_Bytes

#Erase every valye
def Clean(TestEntity):
    TestEntity.ApplyAlpha_Pixel = 0
    TestEntity.ApplyMask = 0
    TestEntity.CLUT_Pixel_Applied = 0
    TestEntity.RAM_put = 0
    TestEntity.RAM_get = 0
    TestEntity.RAM_Used = 0
    TestEntity.RAM_Used_Bytes = 0

#Analyze by printing histogram of each array
def Analyze(TestEntity):
    histogram(TestEntity.ApplyAlpha_Pixel_Array)
    histogram(TestEntity.ApplyMask_Pixel_Array)
    histogram(TestEntity.CLUT_Pixel_Array)
    histogram(TestEntity.RAM_put_Array)
    histogram(TestEntity.RAM_get_Array)
    histogram(TestEntity.RAM_Used_Bytes_Array)
```

1.4 CLUT.py

```
#Class for color lookup table with R, G and B channels
import numpy as np


def GenerateCLUT(sizeR, sizeG, sizeB):
    #Generate a color lookup table for R, G and B channels with each channel having a si
    CLUTR = np.zeros((1, sizeR), dtype=np.uint8)
    CLUTG = np.zeros((1, sizeG), dtype=np.uint8)
    CLUTB = np.zeros((1, sizeB), dtype=np.uint8)
    #Merge CLUTR, CLUTG and CLUTB into a single 2D array called CLUT
    CLUT = np.concatenate((CLUTR, CLUTG, CLUTB), axis=0)
    return CLUT



def GenerateTestCLUT(sizeR, sizeG, sizeB):
    #Generate a color lookup table for R, G and B channels with each channel having a si
    #Only similar CLUT size-values atm
    CLUT = np.zeros((3, sizeR), dtype=np.uint8)
    #Generate the CLUT as 2D arrays in R, G, B order.
    for i in range(sizeR):
        CLUT[0][i] = sizeR - i
        CLUT[1][i] = sizeG - i
        CLUT[2][i] = sizeB - i
    return CLUT


def ChangeCLUT(OldCLUT, NewCLUT, TestEntity):
        #replace OldCLUT with NewCLUT. OldCLUT and NewCLUT are 2D arrays that contain th
        #OldCLUT and NewCLUT are of the same bit-size.
        OutCLUT = [[[0 for b in range (len(NewCLUT[0][0]))] for g in range (len(NewCLUT[

        for i in range (len(OldCLUT)):
            for CurrentX in range (len(OldCLUT[i])):
                OutCLUT[i][CurrentX] = NewCLUT[i][CurrentX]
        TestEntity.ChangeCLUT += 1
        return OutCLUT


def ApplyCLUT(Picture, CLUT, X_Offset, TestEntity):

    PictureOut = np.zeros((800, 4), dtype=np.uint8)
    #Over the range of picture
    for CurrentX in range(len(Picture)):
        if TestEntity.FreeLine[CurrentX+X_Offset]:


            #Temp[0], Temp[1], Temp[2] = CLUT[0][Picture[CurrentX][0]], CLUT[1][Picture[



            #Set the pixel to the value in the CLUT
            PictureOut[CurrentX+X_Offset][0], PictureOut[CurrentX+X_Offset][1], PictureO
            if (len(Picture[CurrentX]) == 4):
                PictureOut[CurrentX+X_Offset][3] = Picture[CurrentX+X_Offset][3]

            if TestEntity.CurrentLayer != 0:
                TestEntity.FreeLine[CurrentX + X_Offset] = False
            TestEntity.CLUT_Pixel_Applied += 1







    TestEntity.CLUT_Applied += 1

    return PictureOut
```

1.5 DemoScene.py

```
import AlphaBlending
import AlphaMasking
import OperationsCounter
import Dynamic_RAM
import CLUT
from PIL import Image
import Analytics
import numpy as np
import MenusConstructor

import time

#If you want to input custom values. Testing grounds.
""" print("Custom resolution? Yes/No")
if(input()==("Yes" or "yes")):
    print("Please input X resolution")
    ScreenResolutionX = input()
    print("Please input Y resolution")
    ScreenResolutionY = input()
else:
    print("Okay, No custom resolution :( Setting to standard resolution 800x600")
    ScreenResolutionX = 800
    ScreenResolutionY = 600 """

#Get screen sizes
ScreenResolutionX = MenusConstructor.ScreenResolutionX
```

```
ScreenResolutionY = MenusConstructor.ScreenResolutionY

#Setup
TestEntity = OperationsCounter.OperationsCounter(ScreenResolutionY, ScreenResolutionX)
StateMachineStatus = "Main"
OutputBuffer = np.zeros((ScreenResolutionX, 4), dtype=np.uint8)
OutputBufferLarge = np.zeros((ScreenResolutionY, ScreenResolutionX, 4), dtype=np.uint8)

#Create RAM
RAM=Dynamic_RAM.RAM(16, TestEntity)
TestWithTime = True



#Main render function, where the drawing magic happens.
def Render(Items, CurrentY, RAM, TestEntity):
    TestEntity.FreeLine = [True]*ScreenResolutionX
    #Create a linebuffer for shared use. Will always get position 0.
    LineBufferAdress = RAM.put(np.zeros((ScreenResolutionX, 4), dtype=np.uint8), TestEnt

    CurrentItem = Items[len(Items)-1]
    #Start at top layer. Iterate downwards towards background.
    while CurrentItem.Next != None:
        TestEntity.CurrentLayer = CurrentItem.Layer


        RAM.StorePictureInRam(CurrentItem, CurrentY, RAM, TestEntity)

        LineBuffer = RAM.get(LineBufferAdress, TestEntity)
        #If current layer is within current y, proceed to do operations
        if ((CurrentY >= CurrentItem.PictureOffset[1]) and (CurrentY < CurrentItem.Pictu




            if(CurrentItem.ApplyCLUT):
                    CLUTBuff = CLUT.ApplyCLUT(RAM.get(CurrentItem.Picture_RAM_Adress, Te

                    # fill buffer with used pixels
                    for x in range (ScreenResolutionX):
                        if TestEntity.FreeLine[x] == False:
                            LineBuffer[x] = CLUTBuff[x]






            if(CurrentItem.ApplyMask == True and CurrentItem.ApplyAlpha == False):


                    MaskBuff = AlphaMasking.ApplyMask(RAM.get(CurrentItem.Picture_RAM_Ad

                    # fill buffer with used pixels
                    for x in range (ScreenResolutionX):
                        if TestEntity.FreeLine[x] == False:
                            LineBuffer[x] = MaskBuff[x]
            if(CurrentItem.ApplyAlpha):

                    #workaround to use both alpha and mask
                    if(CurrentItem.ApplyMask):
                    AlphaBuff = AlphaMasking.ApplyMask(RAM.get(CurrentItem.Picture_RAM_A

                    Temp = RAM.get(CurrentItem.Picture_RAM_Adress, TestEntity)

                    for x in range (len(Temp)):
                        if TestEntity.FreeLine[x+CurrentItem.PictureOffset[0]] == False:
                            Temp[x] = AlphaBuff[x + CurrentItem.PictureOffset[0]]
                    RAM.put_specific(CurrentItem.Picture_RAM_Adress, Temp, TestEntity)

                    #real alpha part
                    else:
                    AlphaBuff = AlphaBlending.ApplyAlpha(RAM.get(CurrentItem.Picture_RAM

                    for x in range (ScreenResolutionX):
                        if TestEntity.FreeLine[x] == False:
                            LineBuffer[x] = AlphaBuff[x]




        #Store in ram
        RAM.put_specific(LineBufferAdress, LineBuffer, TestEntity)

        #Move to next layer
        CurrentItem = CurrentItem.Next


    #Draw background when at end of array
    CurrentItem = Items[1]
    TestEntity.CurrentLayer = CurrentItem.Layer


    RAM.StorePictureInRam(CurrentItem, CurrentY, RAM, TestEntity)
    #Possibly apply CLUT if requested
    if(CurrentItem.ApplyCLUT):

        Buff = CLUT.ApplyCLUT(RAM.get(CurrentItem.Picture_RAM_Adress, TestEntity), Curre
        Temp = RAM.get(CurrentItem.Picture_RAM_Adress, TestEntity)
        for x in range (ScreenResolutionX):
            #Knotete INC.
            Temp[x] = Buff[x]
        RAM.put_specific(CurrentItem.Picture_RAM_Adress, Temp, TestEntity)




    #Fill rest of unmodified pixels with background
```

```
    for x in range(ScreenResolutionX):
        if(TestEntity.FreeLine[x] == True):
            LineBuffer[x] = Items[1].Picture[CurrentY][x]





    return RAM.get(LineBufferAdress, TestEntity), TestEntity






Runs = 0
while(1):


    #Construct the main scene if statemachine requests it
    if (StateMachineStatus == "Main"):

        #Clean testing
        Analytics.Clean(TestEntity)

        #Construct the MainMenu Scene Descriptor list
        MainMenu = MenusConstructor.MainMenuBuild(TestEntity)

        #Start timing
        TestWithTime = True
        #Total time
        MuskTime = time.time()
        for CurrentY in range(ScreenResolutionY):
            TestEntity.CurrentY = [CurrentY]

            #MainLoop for render
            #Line-time
            StartTime = time.time()
            #Start render
            OutputBuffer, TestEntity = Render(MainMenu, CurrentY, RAM, TestEntity)
            EndTime = time.time()

            RAM.CheckEveryArrayBit(TestEntity)
            #Cleanup
            RAM.clear("All", TestEntity)
            #End MainLoop
            #Save time
            TestEntity.TimeTaken[CurrentY] = EndTime - StartTime
            #OutputBufferLarge is only used to visualize the whole picture at the end. N
            OutputBufferLarge[CurrentY] = OutputBuffer


            #More analytics
            Analytics.Testing(TestEntity, CurrentY)
            Analytics.Clean(TestEntity)

        MoskTime = time.time()
        FullTime = MoskTime - MuskTime
        print(FullTime)



        #Print TestEntity into Histograms
        Analytics.histogram(TestEntity.TimeTaken)
        Analytics.Analyze(TestEntity)

        #draw OutputBufferLarge to screen, and save it locally
        OutputBufferPicture = Image.fromarray(OutputBufferLarge)
        OutputBufferPicture.save("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Billed
        #OutputBufferPicture.show()
        print("PrintedPicture")

        #Run MainMenu for x cycles
        Runs += 1
        del MainMenu
        if (Runs >= 1):
            StateMachineStatus = "SettingsMenu"
            Runs = 0
            #Hmm


        #Acts the same way as MainMenu when ran. No comments neccesary.
    if (StateMachineStatus == "SettingsMenu"):
        Analytics.Clean(TestEntity)

        SettingsMenu = MenusConstructor.SettingsMenuBuild(TestEntity)

        TestWithTime = True
        for CurrentY in range(ScreenResolutionY):
            TestEntity.CurrentY = [CurrentY]


            StartTime = time.time()
            OutputBuffer, TestEntity = Render(SettingsMenu, CurrentY, RAM, TestEntity)
            EndTime = time.time()
            RAM.CheckEveryArrayBit(TestEntity)
            RAM.clear("All", TestEntity)


            Analytics.Testing(TestEntity, CurrentY)
            Analytics.Clean(TestEntity)


            OutputBufferLarge[CurrentY] = OutputBuffer



        Analytics.histogram(TestEntity.TimeTaken)
        Analytics.Analyze(TestEntity)
    OutputBufferPicture = Image.fromarray(OutputBufferLarge)
    OutputBufferPicture.save("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Billed

    Runs += 1
    del SettingsMenu
    if (Runs >= 1):
        StateMachineStatus = "SubSettingsMenu"
        Runs = 0
        #Hmm

    #No comments needed
if (StateMachineStatus == "SubSettingsMenu"):
    Analytics.Clean(TestEntity)


    SubSettingsMenu = MenusConstructor.SubSettingsMenuBuild(TestEntity)

    TestWithTime = True
    for CurrentY in range(ScreenResolutionY):
        TestEntity.CurrentY = [CurrentY]

        #MainLoop for render
        StartTime = time.time()
        OutputBuffer, TestEntity = Render(SubSettingsMenu, CurrentY, RAM, TestEntity
        EndTime = time.time()
        RAM.CheckEveryArrayBit(TestEntity)
        RAM.clear("All", TestEntity)
        #End MainLoop

        TestEntity.TimeTaken[CurrentY] = EndTime - StartTime

        Analytics.Testing(TestEntity, CurrentY)

        Analytics.Clean(TestEntity)

        OutputBufferLarge[CurrentY] = OutputBuffer


    Analytics.histogram(TestEntity.TimeTaken)
    Analytics.Analyze(TestEntity)


    OutputBufferPicture = Image.fromarray(OutputBufferLarge)
    OutputBufferPicture.save("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Billed

    Runs += 1
    del SubSettingsMenu
    if (Runs >= 1):
        StateMachineStatus = "Done"
        Runs = 0
        #Hmm

else:
    break
```

1.6 Dynamic$_R$*AM.py*

```
#This code will simulate a dynamic RAM module with a set size as init input argument.
#It will increment an attribute of the input class "TestEntity" every time each function


from numpy import true_divide

#Initiate a RAM module
class RAM:
    def __init__(self, Size, TestEntityIn):
        self.Size = Size
        self.data = [None] * Size
        self.dataoccupied = [False] * Size
        self.RAMTestEntity = TestEntityIn


# A function to put data into the RAM array, and return an andress
    def put(self, data, TestEntity):
        TestEntity.RAM_put_call += 1

        #loop through dataoccupied to see if there is space left in the ram. If there is
        for i in range(len(self.dataoccupied)):
            if self.dataoccupied[i] == False:
                self.data[i] = data
                self.dataoccupied[i] = True
                TestEntity.RAM_put += 1
                return i
        #If no place left in the ram, print error.
        print("No space left in the RAM")
        return -1

    #Put data in specific RAM adress. Similar to regular put, but no scanning needed.
    def put_specific(self, adress, data, TestEntity):
        self.data[adress] = data
        self.dataoccupied[adress] = True
        TestEntity.RAM_put += 1
        return adress


    #Function to get the data at a given adress
    def get(self, adress, TestEntity):
        TestEntity.RAM_get_call += 1

        #check if the adress is in the ram
        #If not in RAM
        if self.dataoccupied[adress] == False:
            print("Data not in the RAM")
            TestEntity.RAM_get_DataNotFound += 1
            return -1

        #If in RAM
        elif self.dataoccupied[adress] == True:
            TestEntity.RAM_get += 1
            return self.data[adress]
        #Situations not covered will give an error.
        elif (self.dataoccupied[adress] == True and self.data[adress] == None) or (self.
            TestEntity.RAM_get_error += 1
            print("Fatal Error. Situation not covered by the code.")
            return -1


#Function to clear out the ram of either all data, or select data.
    def clear(self, adress, TestEntity):
        if adress == "All":
            for i in range(len(self.dataoccupied)):
                self.data[i] = None
                self.dataoccupied[i] = False
                TestEntity.RAM_clear += 1
            return 0
        else:
            self.data[adress] = None
            self.dataoccupied[adress] = False
            TestEntity.RAM_clear += 1

            #If commando "All", clear all data in the ram.

            return True


#Function to check how many "entities" are in the RAM
    def check(self, TestEntity):
        count = 0
        for i in range(len(self.dataoccupied)):
            if self.dataoccupied[i] == True:
                count += 1
        #print("There are", count, "data in the RAM")
        #TestEntity.RAM_check += 1
        return count


   #Function to find how many bytes are stored in RAM
    def CheckEveryArrayBit(self, TestEntity):
        DataSize = 0
        DataArray = 0
        #DataBits = 0
        for i in range(len(self.data)):
            if (self.dataoccupied[i]):
                X = len(self.data[i]) - 1
                Z = len(self.data[i][X])

                DataArray =  X * Z
            else:
                break

            TestEntity.RAM_Used_Bytes +=  DataArray

        #print("There are", DataSize, "databits in the RAM")
        return
```

```
#Store a Y-axis of a picture in ram
    def StorePictureInRam(self, Structure, CurrentY, RAM_Entity, TestEntity):
        #Save the current line of the active picture in the RAM.
        #
        if (CurrentY < Structure.PictureSize[1] + Structure.PictureOffset[1]) and (Curren
            Structure.Picture_RAM_Adress = RAM_Entity.put(Structure.Picture[CurrentY-Str
            Structure.PictureStoredInRam = True
        else:
            Structure.PictureStoredInRam = False
```

1.7 MenusCOnstructor.py

```
import Scene_Descriptor
import AlphaBlending
import numpy as np
from PIL import Image
import CLUT

#Setup
ScreenResolutionX = 800
ScreenResolutionY = 600


#Build Info for MainMenu
#Tips: always include mask for text. Match the mask with a blackout of the text.
def MainMenuBuild(TestEntity):
    #Use Scene_Descriptor to build the scene through a function
    MainMenuBG = Scene_Descriptor.SceneDescriptor(0, ScreenResolutionX, ScreenResolution

    MainMenuBG.Exists = 1
    #MainMenu1.Layer = 0



    #set background to screen resolution
    #Save picture and CLUT in the class
    MainMenuBG.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Bi
    MainMenuBG.Picture = np.asarray(MainMenuBG.Picture.convert('RGBA'))
    MainMenuBG.PictureSize = [ScreenResolutionX, ScreenResolutionY]
    MainMenuBG.ApplyCLUT = False
    MainMenuBG.CLUT = CLUT.GenerateTestCLUT(256, 256, 256)
    #MainMenuBG.DrawOverBG = False

    #Background within defined screensize?
    if ((MainMenuBG.PictureSize[1] <= ScreenResolutionY) and (MainMenuBG.PictureSize[0])
        pass
    else:
        print("Error! Picture is not the same size as the screen resolution. The backgrou
        exit()

    #Construct layer 1
    MainMenu1 = Scene_Descriptor.SceneDescriptor(1, 200, 100)
    MainMenu1.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Bil
    MainMenu1.Picture = MainMenu1.Picture.convert('RGBA')
    MainMenu1.Picture = AlphaBlending.PutAlpha(np.asarray(MainMenu1.Picture), 200)
    MainMenu1.PictureSize = [200, 100]
```

```
MainMenu1.PictureOffset = [600, 0]
MainMenu1.CLUT = CLUT.GenerateTestCLUT(256, 256, 256)


#MainMenu1.ApplyCLUT = True
MainMenu1.ApplyMask = True
MainMenu1.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Ma


#Construct layer 2
MainMenu2 = Scene_Descriptor.SceneDescriptor(2, 100, 100)
MainMenu2.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Bil
MainMenu2.Picture = MainMenu2.Picture.convert('RGBA')
MainMenu2.Picture = AlphaBlending.PutAlpha(np.asarray(MainMenu2.Picture), 255)
MainMenu2.PictureSize = [100, 100]


MainMenu2.ApplyMask = True
MainMenu2.PictureOffset = [700, 300]
MainMenu2.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Ma



#Construct layer 3
MainMenu3 = Scene_Descriptor.SceneDescriptor(3, 300, 100)
MainMenu3.PictureSize = [300, 100]

MainMenu3.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Bil
MainMenu3.Picture = MainMenu3.Picture.convert('RGBA')
MainMenu3.Picture = AlphaBlending.PutAlpha(np.asarray(MainMenu3.Picture), 120)

MainMenu3.PictureOffset = [0, 200]
MainMenu3.ApplyAlpha = True
MainMenu3.Picture = AlphaBlending.PutAlpha(MainMenu3.Picture, 120)

#MainMenu3.ApplyCLUT = True
#MainMenu3.CLUT = CLUT.GenerateTestCLUT(256, 256, 256)
#End-entity
MainMenuEnd = Scene_Descriptor.Empty

#Create scene array
MainMenu = Scene_Descriptor.SceneItems
MainMenu.Array = [MainMenuBG, MainMenu1, MainMenu2, MainMenu3]

#Create a linked list
MainMenu3.Next = MainMenu2
MainMenu2.Next = MainMenu1
MainMenu1.Next = MainMenuEnd
#MainMenuBG.Next = MainMenuEnd
MainMenuEnd.Next = None

TestEntity.ScenesConstructed += 1
return MainMenu, MainMenuBG, MainMenu1, MainMenu2, MainMenu3
```

```
#The following scene constructions should need no more explanation.



MainMenuBG = Scene_Descriptor.SceneDescriptor(0, ScreenResolutionX, ScreenResolutionY)

#Construct Settings Menu
#Tips: always include mask for text. Match the mask with a blackout of the text.
def SettingsMenuBuild(TestEntity):
    SettingsMenuBuild = Scene_Descriptor.SceneDescriptor(0, ScreenResolutionX, ScreenRes

    SettingsMenuBuild.Exists = 1

    #set background to screen resolution
    SettingsMenuBuild.Picture = Image.open(”F:/Google Drive/Skule/Elsys 5.  r /Nordic Ma
    SettingsMenuBuild.Picture = np.asarray(SettingsMenuBuild.Picture.convert(’RGBA’))
    SettingsMenuBuild.PictureSize = [ScreenResolutionX, ScreenResolutionY]
    SettingsMenuBuild.ApplyCLUT = False
    SettingsMenuBuild.CLUT = CLUT.GenerateTestCLUT(256, 256, 256)



    if ((SettingsMenuBuild.PictureSize[1] <= ScreenResolutionY) and (SettingsMenuBuild.P
        pass
    else:
        print(”Error! Picture is not the same size as the screen resolution. The backgrou
        exit()

    #Buffer Picture 1
    SettingsMenuBuild1 = Scene_Descriptor.SceneDescriptor(1, 400, 200)
    SettingsMenuBuild1.Picture = Image.open(”F:/Google Drive/Skule/Elsys 5.  r /Nordic M
    SettingsMenuBuild1.Picture = SettingsMenuBuild1.Picture.convert(’RGBA’)
    SettingsMenuBuild1.Picture = AlphaBlending.PutAlpha(np.asarray(SettingsMenuBuild1.Pi
    SettingsMenuBuild1.PictureOffset = [0, 100]
    SettingsMenuBuild1.ApplyCLUT = False
    SettingsMenuBuild1.CLUT = CLUT.GenerateTestCLUT(256, 256, 256)


    #MainMenu1.ApplyCLUT = True
    SettingsMenuBuild1.ApplyMask = True
    SettingsMenuBuild1.Mask = np.asarray(Image.open(”F:/Google Drive/Skule/Elsys 5.  r /


    #Buffer Picture 2
    SettingsMenuBuild2 = Scene_Descriptor.SceneDescriptor(2, 200, 100)
    SettingsMenuBuild2.Picture = Image.open(”F:/Google Drive/Skule/Elsys 5.  r /Nordic M
    SettingsMenuBuild2.Picture = SettingsMenuBuild2.Picture.convert(’RGBA’)
    SettingsMenuBuild2.Picture = AlphaBlending.PutAlpha(np.asarray(SettingsMenuBuild2.Pi
    SettingsMenuBuild2.ApplyMask = True
    SettingsMenuBuild2.Mask = np.asarray(Image.open(”F:/Google Drive/Skule/Elsys 5.  r /

    SettingsMenuBuild2.ApplyAlpha = True

    SettingsMenuBuild2.PictureOffset = [300, 500]
    #Buffer Picture 3
    SettingsMenuBuild3 = Scene_Descriptor.SceneDescriptor(3, 300, 100)
    SettingsMenuBuild3.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic M
    SettingsMenuBuild3.Picture = SettingsMenuBuild3.Picture.convert('RGBA')
    SettingsMenuBuild3.Picture = AlphaBlending.PutAlpha(np.asarray(SettingsMenuBuild3.Pi
    SettingsMenuBuild3.PictureOffset = [500, 500]

    SettingsMenuBuild3.ApplyMask = True
    SettingsMenuBuild3.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /

    #Buffer Picture 4
    SettingsMenuBuild4 = Scene_Descriptor.SceneDescriptor(4, 200, 100)
    SettingsMenuBuild4.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic M
    SettingsMenuBuild4.Picture = SettingsMenuBuild4.Picture.convert('RGBA')
    SettingsMenuBuild4.Picture = AlphaBlending.PutAlpha(np.asarray(SettingsMenuBuild4.Pi
    SettingsMenuBuild4.PictureOffset = [500, 100]

    SettingsMenuBuild4.ApplyMask = True
    SettingsMenuBuild4.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /

    SettingsMenuBuildEnd = Scene_Descriptor.Empty

    SettingsMenu = Scene_Descriptor.SceneItems
    SettingsMenu.Array = [SettingsMenuBuild, SettingsMenuBuild1, SettingsMenuBuild2, Sett

    #Create a linked list
    SettingsMenuBuild4.Next = SettingsMenuBuild3
    SettingsMenuBuild3.Next = SettingsMenuBuild2
    SettingsMenuBuild2.Next = SettingsMenuBuild1
    SettingsMenuBuild1.Next = SettingsMenuBuildEnd
    #SettingsMenuBuild.Next = SettingsMenuBuildEnd
    SettingsMenuBuildEnd.Next = None

    TestEntity.ScenesConstructed += 1

    SettingsMenu = Scene_Descriptor.SceneItems
    SettingsMenu.Array = [SettingsMenuBuild, SettingsMenuBuild1, SettingsMenuBuild2, Sett

    return SettingsMenu, SettingsMenuBuild, SettingsMenuBuild1, SettingsMenuBuild2, Setti







#Construct Subsettings Menu
def SubSettingsMenuBuild(TestEntity):

    SubSettingsMenuBuild = Scene_Descriptor.SceneDescriptor(0, ScreenResolutionX, Screen

    SubSettingsMenuBuild.Exists = 1
    #SubSettingsMenu.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Mas
    SubSettingsMenuBuild.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic
    SubSettingsMenuBuild.Picture = np.asarray(SubSettingsMenuBuild.Picture.convert('RGBA
    SubSettingsMenuBuild.PictureSize = [ScreenResolutionX, ScreenResolutionY]

    SubSettingsMenuBuild.ApplyCLUT = True
    SubSettingsMenuBuild.CLUT = CLUT.GenerateTestCLUT(256, 256, 256)


    SubSettingsMenuBuild1 = Scene_Descriptor.SceneDescriptor(1, 800, 600)
    SubSettingsMenuBuild1.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordi
    SubSettingsMenuBuild1.Picture = SubSettingsMenuBuild1.Picture.convert('RGBA')
    SubSettingsMenuBuild1.Picture = AlphaBlending.PutAlpha(np.asarray(SubSettingsMenuBui

    SubSettingsMenuBuild1.ApplyMask = True
    SubSettingsMenuBuild1.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.

    SubSettingsMenuBuild2 = Scene_Descriptor.SceneDescriptor(2, 800, 600)
    SubSettingsMenuBuild2.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordi
    SubSettingsMenuBuild2.Picture = SubSettingsMenuBuild2.Picture.convert('RGBA')
    SubSettingsMenuBuild2.Picture = AlphaBlending.PutAlpha(np.asarray(SubSettingsMenuBui

    SubSettingsMenuBuild2.ApplyMask = True
    SubSettingsMenuBuild2.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.



    SubSettingsMenuBuild3 = Scene_Descriptor.SceneDescriptor(3, 400, 200)
    SubSettingsMenuBuild3.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordi
    SubSettingsMenuBuild3.Picture = SubSettingsMenuBuild3.Picture.convert('RGBA')
    SubSettingsMenuBuild3.Picture = AlphaBlending.PutAlpha(np.asarray(SubSettingsMenuBui

    SubSettingsMenuBuild3.PictureOffset = [int(ScreenResolutionX/2 - 200), int(ScreenRes
    SubSettingsMenuBuild3.ApplyMask = True
    SubSettingsMenuBuild3.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.

    SettingsMenu = Scene_Descriptor.SceneItems
    SettingsMenu.Array = [SubSettingsMenuBuild, SubSettingsMenuBuild1, SubSettingsMenuBu


    SubSettingsMenuBuildEnd = Scene_Descriptor.Empty
    #Create a linked list
    SubSettingsMenuBuild3.Next = SubSettingsMenuBuild2
    SubSettingsMenuBuild2.Next = SubSettingsMenuBuild1
    SubSettingsMenuBuild1.Next = SubSettingsMenuBuildEnd
    #SettingsMenuBuild.Next = SettingsMenuBuildEnd
    SubSettingsMenuBuildEnd.Next = None

    TestEntity.ScenesConstructed += 1

    SubSettingsMenu = Scene_Descriptor.SceneItems
    SubSettingsMenu.Array = [SubSettingsMenuBuild, SubSettingsMenuBuild1, SubSettingsMen

    return SubSettingsMenu, SubSettingsMenuBuild, SubSettingsMenuBuild1, SubSettingsMenu
```

1.8 OperationsCounter.py

```
#A test class with attributes that measures times a function has been used.

#Length is the y-length of the screen.

#Just a lot of values for reading different values
class OperationsCounter():
    def __init__(self, Length, Width):

        FreeLine = [True]*Width
        self.Length = Length

        self.CurrentLayer = 0

        self.ScenesConstructed = 0
        self.TimeTaken =[0]*Length
        #Alpha
        #Redundant v

        #Alpha
        self.AlphaPassed = 0

        self.ApplyAlpha_Pixel = 0
        self.ApplyAlpha_Pixel_Array = [0]*Length
        self.ApplyAlphaR = 0
        self.ApplyAlphaG = 0
        self.ApplyAlphaB = 0

        self.AlphaBlend = 0

        #Mask
        self.NoMask = 0
        self.ApplyMask = 0
        self.ApplyMask_Pixel_Array = [0]*Length

        #CLUT
        self.ChangeCLUT = 0
        self.CLUT_Applied = 0
        self.CLUT_Pixel_Applied = 0
        self.CLUT_Pixel_Array = [0]*Length
        self.CLUT_Pixel_Skipped = 0

        #RAM
        self.RAM_put_call = 0
        self.RAM_put = 0
        self.RAM_put_Array = [0]*Length

        self.RAM_get_call = 0
        self.RAM_get = 0
        self.RAM_get_Array = [0]*Length

        self.RAM_get_DataNotFound = 0
        self.RAM_get_error = 0
        self.RAM_clear = 0
        self.RAM_Used_Cumulative = 0
        self.RAM_Used_Bits_Cumulative = 0
        self.RAM_Used = 0
        self.RAM_Used_Array = [0]*Length
        self.RAM_Used_Bytes = 0
        self.RAM_Used_Bytes_Array = [0]*Length

        #self.list =

    """  @property
    def NoMask(self):
        return self._NoMask
    @NoMask.setter
    def NoMask(self, value):
        self._NoMask = value
        print("value: ")
        print(self._NoMask) """
```

1.9 Referanse_DobbelBuffer_Enkel.py

```
import OperationsCounter
import Dynamic_RAM
import CLUT
from PIL import Image
import Analytics
import numpy as np
import Referanse_Funksjoner

import time

#Simple scene descriptor lite, based on model scene descriptor
class PictureList:
    def __init__(Self, Layers):
        Self.list = [0]*Layers

class Picture:
    def __init__(Self, layer, ResolutionX, ResolutionY):
        Self.Exists = 1
        Self.Size = [ResolutionX, ResolutionY]
        Self.Layer = layer
        Self.Picture = [[]]
        Self.Offset = [0, 0]
        Self.ApplyCLUT = False
        Self.CLUT = np.zeros((3, 256), dtype=np.uint8)
        Self.ApplyMask = False
        Self.Mask = [[]]
        Self.ApplyAlpha = False



ScreenResolutionX = 800
ScreenResolutionY = 600


TargetBuffer = False
TestEntity = OperationsCounter.OperationsCounter(ScreenResolutionX, ScreenResolutionY)
StateMachineStatus = "MainMenu"
RAM = Dynamic_RAM.RAM(16, TestEntity)
```

```
#A lot of manual construction of scenes, using the scene descriptor.

#Main Menu
MainMenuBG = Picture(0, ScreenResolutionX, ScreenResolutionY)
MainMenu1 = Picture(1, 200, 100)
MainMenu2 = Picture(2, 100, 100)
MainMenu3 = Picture(3, 300, 100)

MainMenuBG.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Ma
MainMenu1.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Mas
MainMenu2.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Mas
MainMenu3.Picture = Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master/Billeder
MainMenu3.Picture.putalpha(128)
MainMenu3.Picture = np.asarray(MainMenu3.Picture)

MainMenu1.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master
MainMenu2.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Master

MainMenu1.Offset = [600, 0]
MainMenu2.Offset = [700, 300]
MainMenu3.Offset = [0, 200]


#Settings
SettingsMenuBG = Picture(0, ScreenResolutionX, ScreenResolutionY)
SettingsMenu1 = Picture(1, 400, 200)
SettingsMenu2 = Picture(2, 200, 100)
SettingsMenu3 = Picture(3, 300, 100)
SettingsMenu4 = Picture(4, 200, 100)

SettingsMenuBG.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordi
SettingsMenu1.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic
SettingsMenu2.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic
SettingsMenu3.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic
SettingsMenu4.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic

SettingsMenu1.Offset = [0, 100]
SettingsMenu2.Offset = [300, 500]
SettingsMenu3.Offset = [500, 500]
SettingsMenu4.Offset = [500, 100]

SettingsMenu1.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Ma
SettingsMenu2.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Ma
SettingsMenu3.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Ma
SettingsMenu4.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic Ma


#SubSettings
SubSettingsMenuBG = Picture(0, ScreenResolutionX, ScreenResolutionY)
SubSettingsMenu1 = Picture(1, 800, 600)
SubSettingsMenu2 = Picture(2, 800, 600)
SubSettingsMenu3 = Picture(3, 400, 200)

SubSettingsMenuBG.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /No
SubSettingsMenu1.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nor
SubSettingsMenu2.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nor
SubSettingsMenu3.Picture = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nor

SubSettingsMenuBG.CLUT = CLUT.GenerateTestCLUT(256, 256, 256)
SubSettingsMenu1.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic
SubSettingsMenu2.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic
SubSettingsMenu3.Mask = np.asarray(Image.open("F:/Google Drive/Skule/Elsys 5.  r /Nordic

SubSettingsMenuBG.Offset = [0,0]
SubSettingsMenu1.Offset = [0,0]
SubSettingsMenu2.Offset = [0,0]
SubSettingsMenu3.Offset = [int(ScreenResolutionX/2 - 200), int(ScreenResolutionY/2 - 100

TimeArray = [0]*3
while(1):


    if StateMachineStatus == "MainMenu":
        FreeLine=np.full((ScreenResolutionY, ScreenResolutionX), True)


        StartTime = time.time()
        #Operation 1
        Out, FreeLine = Referanse_Funksjoner.Ref_Alpha(MainMenu3, MainMenu3.Offset, MainM
        RAM.clear("All", TestEntity)
        #Operation 2
        Out1, FreeLine = Referanse_Funksjoner.Ref_Mask(MainMenu1.Picture, MainMenu1.Offs
        RAM.clear("All", TestEntity)
        #Operation 3
        Out2, FreeLine = Referanse_Funksjoner.Ref_Mask(MainMenu2.Picture, MainMenu2.Offs
        RAM.clear("All", TestEntity)
        #Operation 4
        Out3 = Referanse_Funksjoner.Ref_Fill(Out2, MainMenuBG.Picture, FreeLine, RAM, Te
        RAM.clear("All", TestEntity)

        #Timing
        EndTime = time.time()
        TimeArray[0] = (EndTime - StartTime)
        #print the time
        #Next state
        StateMachineStatus = "SettingsMenu"

        #Show final output
        #Out3 = Image.fromarray(Out3)
        #Out3.show()

        #Not important, but shows how Shadow buffers switch on a software level.
        if TargetBuffer == True:
            Buffer0 = Out3
        if TargetBuffer == False:
            Buffer1 = Out3
        TargetBuffer != TargetBuffer



    #Rest of scenes are much of the same.
    if StateMachineStatus == "SettingsMenu":
        FreeLine=np.full((ScreenResolutionY, ScreenResolutionX), True)

        StartTime = time.time()
        Out, FreeLine = Referanse_Funksjoner.Ref_Mask(SettingsMenu1.Picture, SettingsMenu
        RAM.clear("All", TestEntity)
        Out1, FreeLine = Referanse_Funksjoner.Ref_Mask(SettingsMenu2.Picture, SettingsMe
        RAM.clear("All", TestEntity)
        Out2, FreeLine = Referanse_Funksjoner.Ref_Mask(SettingsMenu3.Picture, SettingsMe
        RAM.clear("All", TestEntity)
        Out3, FreeLine = Referanse_Funksjoner.Ref_Mask(SettingsMenu4.Picture, SettingsMe
        RAM.clear("All", TestEntity)
        EndTime = time.time()
        TimeArray[1] = (EndTime - StartTime)


        #Out3 = Image.fromarray(Out3)
        #Out3.show()
        StateMachineStatus = "SubSettingsMenus"

    if StateMachineStatus == "SubSettingsMenus":
        FreeLine=np.full((ScreenResolutionY, ScreenResolutionX), True)

        StartTime = time.time()

        Out = Referanse_Funksjoner.Ref_CLUT(SubSettingsMenuBG.Picture, SubSettingsMenuBG
        RAM.clear("All", TestEntity)
        Out1, FreeLine = Referanse_Funksjoner.Ref_Mask(SubSettingsMenu1.Picture, SubSetti
        RAM.clear("All", TestEntity)
        Out2, FreeLine = Referanse_Funksjoner.Ref_Mask(SubSettingsMenu2.Picture, SubSetti
        RAM.clear("All", TestEntity)
        Out3, FreeLine = Referanse_Funksjoner.Ref_Mask(SubSettingsMenu3.Picture, SubSetti
        RAM.clear("All", TestEntity)
        #Out3 = Referanse_Funksjoner.Ref_Fill(Out2, MainMenuBG.Picture, FreeLine, RAM, T


        EndTime = time.time()
        TimeArray[2] = (EndTime - StartTime)

        #Get time
        Analytics.histogram(TimeArray)

        Out3 = Image.fromarray(Out3)
        Out3.show()
        break
```

1.10 Referanse_Funksjoner

```
#Create a function that combines two pictures by using the alpha blending algorithm.
import numpy as np
from PIL import Image


#Create a function that masks the current picture with a mask. Similar to model Mask
def Ref_Mask(Foreground, Offset, Background, Mask, FreeLine, RAM, TestEntity):
    #Store in RAM
    ForegroundAdress = RAM.put(Foreground, TestEntity)
    BackgroundAdress = RAM.put(Background, TestEntity)
    #For Iterate over the whole picture
    for y in range (len(Foreground)):
        for x in range (len(Foreground[y])):
#                   for c in range (len(Foreground[y][x])):
                        #If maskbit empty, do nothing
                        if Mask[y][x].all() == 0:
                            pass
                        #if maskbit set, mask out picture
                        else:
                            Temp1 = RAM.get(ForegroundAdress, TestEntity)
                            Temp2 = RAM.get(BackgroundAdress, TestEntity)
                            Temp2[y+Offset[1]][x+Offset[0]] = Temp1[y][x]
                            RAM.put_specific(BackgroundAdress, Temp2, TestEntity)
                            RAM.put_specific(ForegroundAdress, Temp1, TestEntity)
                            FreeLine[y+Offset[1]][x+Offset[0]] = False

    return RAM.get(BackgroundAdress, TestEntity), FreeLine




#Create a function that applies a CLUT to the current picture. Similar to model CLUT
def Ref_CLUT(Picture, CLUT, Offset, RAM, TestEntity):
    #Store in RAM
    PicAdress = RAM.put(Picture, TestEntity)
    CLUTAdress = RAM.put(CLUT, TestEntity)
    #For whole picture
    for y in range (len(Picture)):
        for x in range (len(Picture[y])):
            #For every channel
                for c in range (len(Picture[y][x])-1):
                    TempPic = RAM.get(PicAdress, TestEntity)
                    TempCLUT = RAM.get(CLUTAdress, TestEntity)
                    #Change colour to accompanied new CLUT value
                    Picture[y][x][c] = CLUT[c][Picture[y][x][c]]
                    RAM.put_specific(PicAdress, Picture, TestEntity)

    return RAM.get(PicAdress, TestEntity)


#Function to apply alpha to a picture. Similar to model Alpha
def Ref_Alpha(StructFG, Offset, StructBG, FreeLine, RAM, TestEntity):
    PictureOut = np.zeros((StructBG.Size[1], StructBG.Size[0], 4), dtype=np.uint8)
    #Store in RAM
    ForegroundAdress = RAM.put(StructFG.Picture, TestEntity)
    BackgroundAdress = RAM.put(StructBG.Picture, TestEntity)
    PictureOutAdress = RAM.put(PictureOut, TestEntity)
    for y in range(len(StructBG.Picture)):
        for x in range(len(StructBG.Picture[0])):
            if x - Offset[0] >= 0 and x - Offset[0] < StructFG.Size[0]:
                if y - Offset[1] >= 0 and y - Offset[1] < StructFG.Size[1]:
                        #If fully transparent
                        if StructFG.Picture[y-Offset[1]][x-Offset[0]][3] == 0:
                            TempFG = RAM.get(ForegroundAdress, TestEntity)
                            TempOut = RAM.get(PictureOutAdress, TestEntity)
                            TempOut[x + Offset[0]] = TempFG[y][x]
                            RAM.put_specific(PictureOutAdress, TempOut, TestEntity)
                            RAM.put_specific(ForegroundAdress, TempFG, TestEntity)
                            TestEntity.AlphaPassed += 1
                        #Blend it
                        #Each channel
                        #Fg*Alpha + Bg*(1-Alpha)
                        else:
                            TempFG = RAM.get(ForegroundAdress, TestEntity)
                            TempBG = RAM.get(BackgroundAdress, TestEntity)
                            TempOut = RAM.get(PictureOutAdress, TestEntity)
                            R = TempFG[y-Offset[1]][x-Offset[0]][0]*TempFG[y-Offset[1]][x-Of
                            TestEntity.ApplyAlphaR += 1
                            B = TempFG[y-Offset[1]][x-Offset[0]][1]*TempFG[y-Offset[1]][x-Of
                            TestEntity.ApplyAlphaG += 1
                            G = TempFG[y-Offset[1]][x-Offset[0]][2]*TempFG[y-Offset[1]][x-Of
                            TestEntity.ApplyAlphaB += 1
                            FreeLine[y][x] = False
                            TempOut[y][x][0], TempOut[y][x][1], TempOut[y][x][2], TempOut[y]
= R, G, B, TempFG[y-Offset[1]][x-Offset[0]][3]
                            RAM.put_specific(PictureOutAdress, TempOut, TestEntity)
                else:
                        TempBG = RAM.get(BackgroundAdress, TestEntity)
                        TempOut = RAM.get(PictureOutAdress, TestEntity)
                        TempOut[y][x] = TempBG[y][x]
                        RAM.put_specific(PictureOutAdress, TempOut, TestEntity)
            else:
                TempBG = RAM.get(BackgroundAdress, TestEntity)
                TempOut = RAM.get(PictureOutAdress, TestEntity)
                TempOut[y][x] = TempBG[y][x]
                RAM.put_specific(PictureOutAdress, TempOut, TestEntity)

        TestEntity.AlphaBlend += 1
    return RAM.get(PictureOutAdress, TestEntity), FreeLine


#Fills any free pixels with background
def Ref_Fill(Picture, Background, FreeLine, RAM, TestEntity):
    PicAdress = RAM.put(Picture, TestEntity)
    BackgroundAdress = RAM.put(Background, TestEntity)
    for y in range(len(Background)):
        for x in range(len(Background[0])):
            if FreeLine[y][x] == True:
                RAM.get(PicAdress, TestEntity)
                RAM.get(BackgroundAdress, TestEntity)
                Picture[y][x] = Background[y][x]
    return Picture
```

1.11 Scene_Descriptor.py

```
#SCENE DESCRIPTOR
import numpy as np
```

```
class Empty():
    def __init__(self):

        self.Next = None


def BuildScene(PictureFG, PictureBG, OperationsCounter):

    return 0


#Array of scenes, for layering purposes
class SceneItems:
    def __init__(self, Items, Size):
        self.Items = []*Items
        self.NumberOfItems = Size
        self.Picture = []

#Class for wrapping scene around picture
class SceneDescriptor:
    #Generate Scene
    def __init__(Self, layer, ResolutionX, ResolutionY):
        Self.Exists = 1
        Self.SizeX = ResolutionX
        Self.SizeY = ResolutionY
        Self.Layer = layer
        Self.DrawOverBG = True

        Self.Picture = [[]]
        Self.PictureSize = [ResolutionX, ResolutionY]
        Self.PictureStoredInRam = False
        Self.Picture_RAM_Adress = 0

        Self.Next = None
        Self.Previous = None

   # class PictureInfo:
        #def __init__(Self):
        #Additional scene operations info
        Self.PictureOffset = [0, 0]


        Self.ApplyAlpha = False
        Self.ApplyTargetAlpha = False
        Self.ApplyAlphaTarget = 0

        Self.ApplyTargetMask = False
        Self.ApplyMaskTarget = 0
        Self.ApplyMask = False
        Self.Mask = [[]]


        Self.ApplyCLUT=False
        Self.CLUT = np.zeros((3, 256), dtype=np.uint8)
        Self.GlobalCLUT=False
Self.GlobalCLUT = np.zeros((3, 256), dtype=np.uint8)

    #X, Y
```