Gymnasium “Svetozar Marković” Niš

Thesis paper

Class: **Informatics**

Topic: **Implementation of an ASCII Shader in Modern Graphics Systems**

Professor-mentor:

Žaklina Eftimovski

Student:

Ristić Leon

January 19, 2024

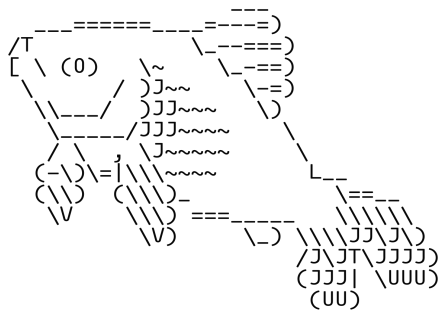
Content

Introduction

Background

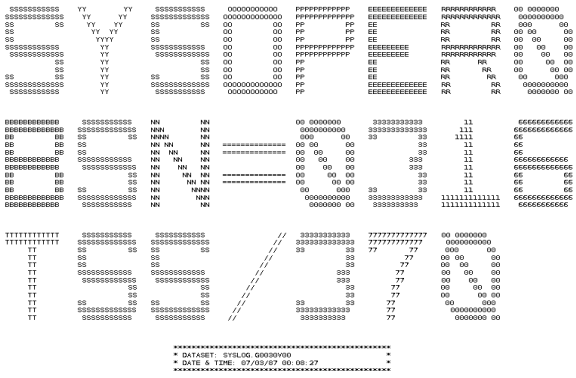
ASCII (American Standard Code for Information Interchange) is a character encoding standard that assigns unique numbers to letters, numbers and symbols, allowing computers to represent and exchange text data. It maps 128 characters, including A-Z, a-z, 0-9, and punctiations, using 7-bit binary codes, forming the fondation for modern character sets like Unicode and many more.

ASCII art on the other hand is a graphic design technique that uses computers for presentation and consists of pictures pieced together from the 95 printable (from a total of 128) characters defined by ACII standard we described earlier.



Simple ASCII art of a fish

ASCII art was invented, in large part, because early printers often lacked graphic ability and thus, characters were used in place of graphic marks. Also, to mark divisions between different print jobs from different users, bulk printers often used ASCII art to print large banner pages, making the division easier to spot so that the results could be more easily separated by a computer operator. ASCII art was also used in early e-mail when images could not be embedded.

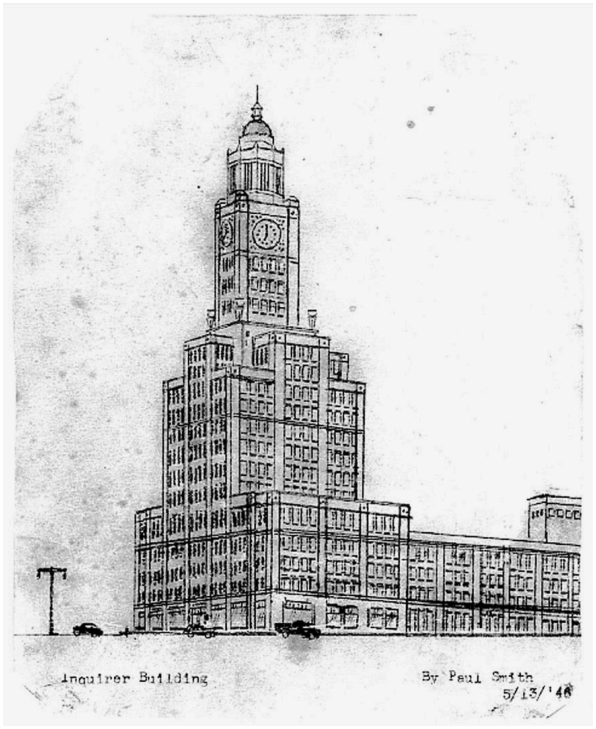


Simple banner page generated by TSS/370

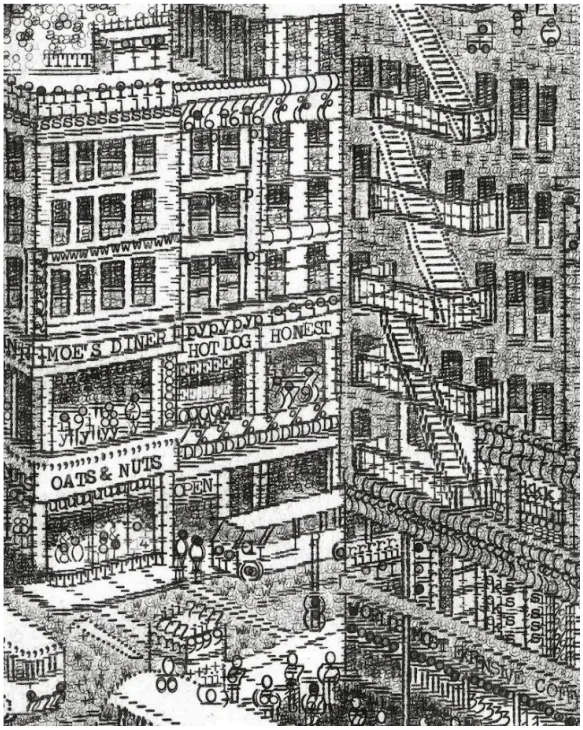
Uses

ASCII art was used where text could be faster and more readable, printable or transmitted then standard graphic images or in some cases where the transmission of images was not possible. These limitations were a big thing for systems like:

- Typewriters;



Inquirer Building by Paul Smith 1946 using manual typewriter;



James Cook is a current-day Briton who expanded on Smith's work.

- Teleprinters;
- Non-graphic computer terminals;

- Printer separators.

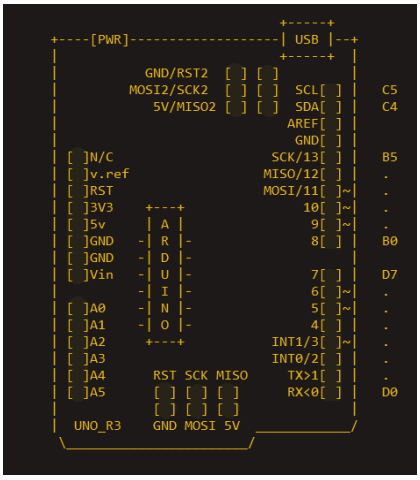
And in early internet days primitive:

- Emails;

- Usenet news messages

After these limitations were overcomed and computer graphics could be imported and displayed the ASCII art stayed as a form of an artistic touch to represent companies and product logos within source codes of computer programs.

Some electronic schematics archives represent the circuits using ASCII art.



As the internet got into every household in western and later in eastern hemisphere of earth more and more people could expand and create with ASCII characters, and as the video games became one of the biggest entertainment fields ASCII art took a global hold on the content creators made and some of

the early games were entirely run on lighter ASCII generated graphics such as *The Kroz* series of rouglelike genres of games made for MS-DOS systems. *Overkill* is also an example of early 2D platform multiplayer shooter game entirely designed with ASCII graphics

The era of gaming was peaking in all time high numbers by the beginning of the 2000s, and many of the culture phenomenons and communities were born in this era.

Without modern video walkthroughs and high speed broadband, the centre of gaming knowledge and guides was *GamingFAQs*. With strict file sizes and constrained dial-up speeds people had to write and distribute these gaming guides exclusively as plain text files (.txt). Lacking the ability to embed screenshots or any sort of graphic, a unique subculture emerged where contributors and authors started using ASCII art to construct logos and headers.

With broad and wide spread of the internet, specific limitations of softwear and hardwear, and pure dedication and talent of the people the conditions were met for ASCII art to become an important cornerstone of early internet culture, demonstrating that human creativity could not be contained with technical constraints creating a unique and enduring diginat aesthetics.

Motivation and Objective

The core objective of this thesis paper is the development of a specialized software environment designed to automate the conversion of raster imagery into stylized, character-based (ASCII) representation of themselves. The motivation for this project is to bridge the gap between high level of proficiency, talent and time requrement for creating ASCII art and high-performance, real-time rendering needed for modern games. In the end we should have working algorithms and methode for replicating ASCII art aesthetics that invokes nostalgic feeling while performing with low demand on hardwear inside modern games.

The project is structured in two distinct development phases. The first phaze (Chapter I) is creating a 2D image editing studio engine inside Pythone in which we will be able to test and easily change our approuch if something isn't right. This 2D implementation serves not only as a standalone image editor, but as a critical proof of concept for the underlying algorithms. By validating our approach in a safe environment we will be able to easly advance to the second part of this project.

The second part (Chapter II) of this project is implementing our approach and algorithm in premade, commercially available softwear called ReShade which is a generic post-processing injector for games and video softwear. At a fundamental level ReShade works as a "man-in-the-middle" for graphics pipeline. When a game application calls upon the Graphics API to render a frame, ReShade intercepts these calls and injects code written in a C-like language called HLSL (high-level Shading Language) to manipulate the pixel data directly on the GPU. Unlike the CPU-based Python implementation, which processes pixels sequentially or in batches, ReShade executes the ASCII conversion we wrote in parallel across thousands of GPU cores which enables real-time rendering and low performance cost on our hardwear. Also an important note and thing for the ReShade that will enable us more graphical fidelity and options is its Z-depth buffer which will be provided by the game's 3D nature and will give as depth-aware ASCII rendering.

Chapter I: The 2D Implementation

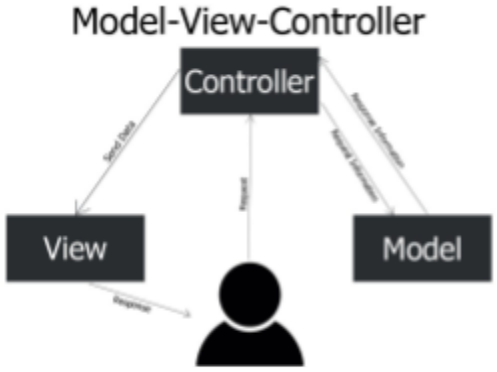
Architectural overview of the "Tilemap Engine"

When I first envisioned this project I imagined this to be user friendly, easily modifiable and available to everybody that wants to try making and generating ASCII art for themself and so the "**Tilemap Engine**" was born. The desktop application is built using Python 3. The architecture follows a modified Model-View-Controller (MVC) pattern to separate the user interface from the rendering logic, simplifying use while being highly consomizable.

The explanation of the Engine architecture could be divided in 4 separate steps for better understanding:

1. User interface

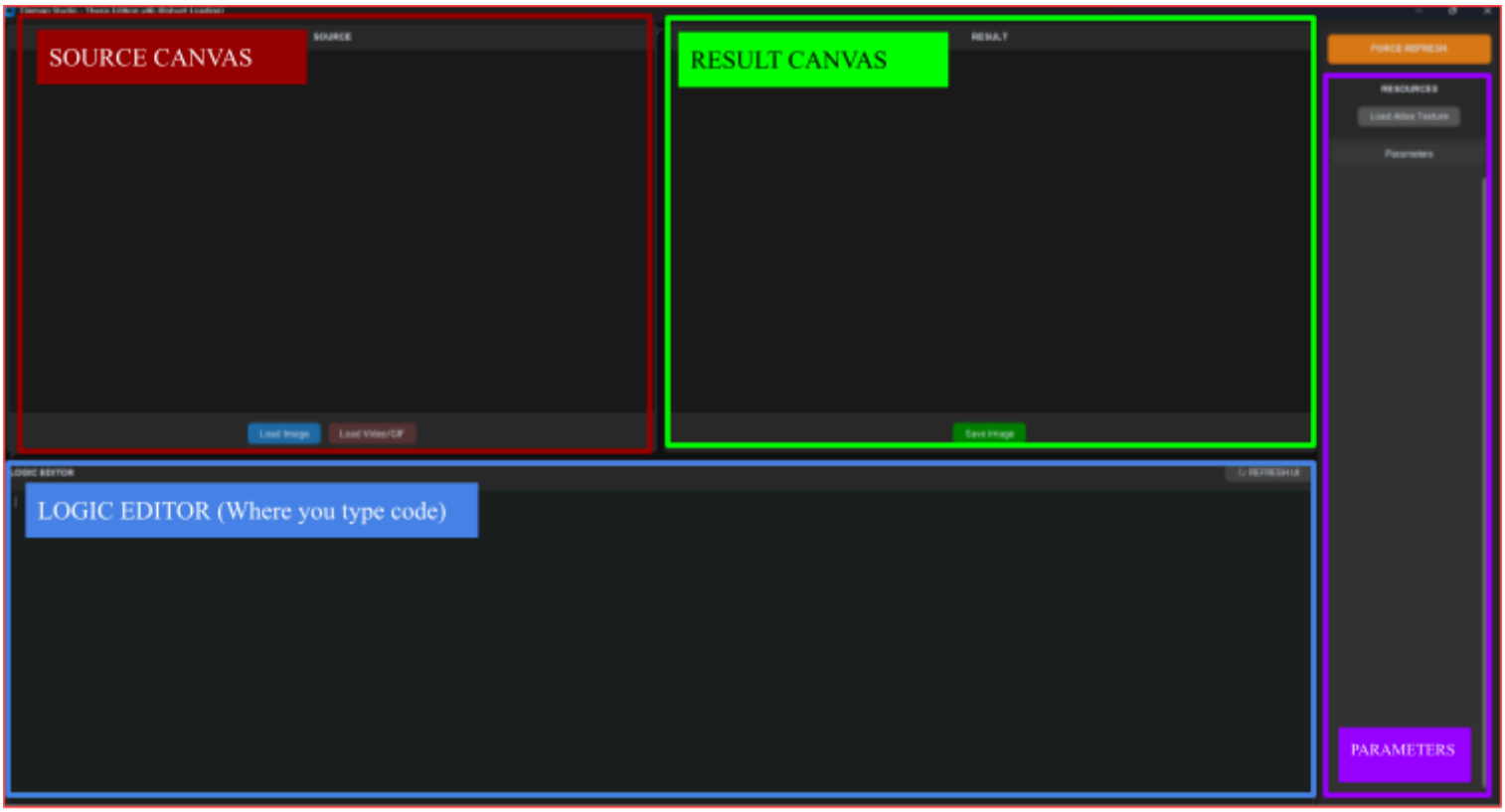
- **CustomTkinter** is used for construction of user friendly interface with modern features such as:
    - Native dark mode support.
    - High-DPI aware widgets (scaling the windows accordingly)
    - Rounded corners and modern aesthetics for buttons and frames.
- **Tkinter** (Standard library) was used only for specific layout management (like PanelWindow for resizable split-views) and the Canvas widget for drawing images.



MVC system described visually.

The application interface (GUI) is divided into three main sections.

1. Visual Workspace (Top)
    - Source Canvas (Left): Shows the loaded image or video. The control bar includes "Load Image" and "Load Video/GIF" buttons to handle file input.
    - Result Canvas (Right): Displays the final ASCII output. The control bar changes automatically and shows "Save Image" for static files or playback controls when a video is active.
2. Logic Editor (Bottom)
    - This text box allows the user to write Python rendering logic directly with no need for retourning to the original code. The "Refresh UI" button scans this text to update the settings on the right.
3. Parameter box (Right Sidebar)
    - Resource Controls: Buttons for "Force Refresh" and "Load Atlas Texture" are always visible at the top, and provide additional control of the results. Atlas Texture is the image file created to store ASCII characters that are going to be drawn later. This allowes for unlimited possibilities for creating all kinds of art.
    - Dynamic Parameters: A scrollable list that automatically creates sliders and checkboxes based on the variables found in the Logic Editor.



User interface and layout used by Tilemap Studio.

2. Image processing and computer vision

- NumPy which was used as the core of the “Vectorized” engine. It handles the mathematical operations (luminescence calculations, gradient detection, array broadcasting) efficiently, replacing slow Python loops.
- OpenCV(cv2) was used for high-performance video encoding since the engine was imagined to render videos and gifs alongside still images. It is using H.264 (avc1) codec for video exports.
- Pillow(PIL) which was used to handle alpha cannels (RGBA) of imported images/videos since not all of them operate in RGB systems and for converting NumPy arrays into formats compatible with the Tkinter UI.

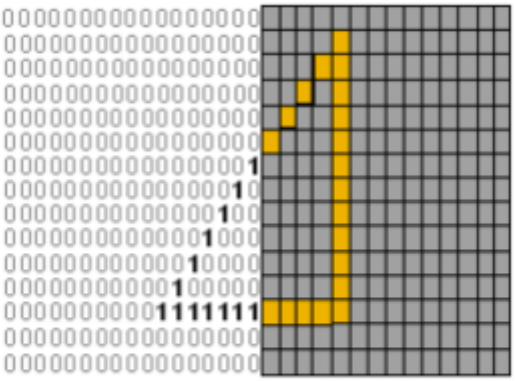
In standard Python if you want to, for example, darken the image, you would write a loop that goes thru every single pixel in the image (for example a full hd image is made of 1920 pixels multiplied by 1080 pixels, so 2073600 pixels in total) and subtracts a RGB value. This is extremely slow because Python has to interpret the subtraction instruction 2 million times.
By using **Vectorization** or also called “SIMDization” which are basically instructions provided by **NumPy** in Python, we are replacing loop commands with instruction directly to the CPU of our machine to process a single element of an array N times, it processes (say) 4 elements of the array simultaneously N/4 times. We are here using NumPy’s Broadcasting capabilities. Instead of treating all 2 million pixels,

NumPy treats it as a single multi-domensional matrix. When we say `image_matrix - 10`, NumPy sends that operation down to the C-level machine code, executing it on all 2 million pixels at the same time (using already mentioned SIMD instructions on the CPU).

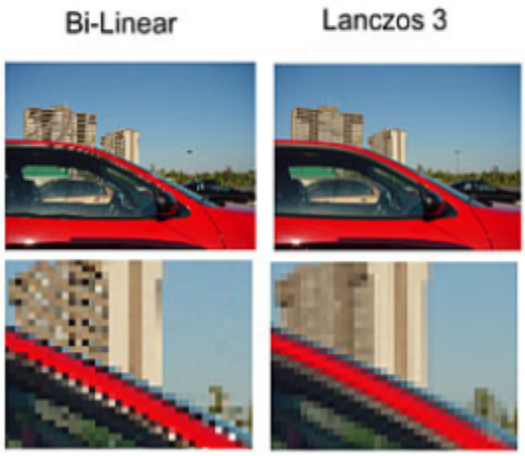
Pillow is the primary library for image processing and manipulation in Python. It is a fork of the standard Python Imaging Library or PIL, which means that some third party took that original codebase (PIL) and started developing it independently. And from the name comes.
Since Windows and Mac operating sistems don't know what raw math information (NumPy arrays and matrices) look like thus being unable to display that directly on the screen; they need a bitmap. Pillow here acts like a translator from data to visualisation.

In the draw function `draw_image_on_canvas` by adding **adaptive resampling** features in two distinct ways we get readable results that are optimized and light on hardwear. There are many ways and algorithms while resampling the images, but I choose these two with my engine with performanse and quality of the final image as deciding factors:

- Lanczos resampling or Lanczos filtering for static images for greater quality since the function `Image.LANCZOS` is heavy for the computing.
- Bilinear interpolation (or resampling) is activated if the video is loaded into the Engine and is playing. This is due to `Image.BILINEAR` being a much lighter function to compute.



Visual explanation of Bitmaps



Difference between Bilinear and Lanczos resampling

This step is a must in the field of ASCII art since the processed images are often downscaled and without resampling, the images would be very small and hard to see.

3. Dynamic Metaprogramming

- AST(Abstract Syntax Tree) is a standard Python library used to parse the code written in the logic box of our Tilemap engine and translate it into raw, but understandable language for the model underneath. It is a sort of a dictionary in which we associate simple commands (etc. TRESHOLD = 10) with more sophistichated commands that actually do the math. Just for the convenience of the engine and usage.

Regarding the complexity of concepts present in this thesis paper, metaprogramming clearly tops the charts. Most times, UI elements (buttons, sliders, checkmarks, etc.) are “hard coded” (e.g., you explicitly write something like `slider = newSlider()`). In “Tilemap engine”, the UI elements build themselves by reading the user's code in the logic box and act correspondingly.
This is done utilizing AST (Abstract Syntax Tree) library inside Python to break the user’s code into a grammatical tree structure (syntax tree) before executing the commands in four steps:

1. Parsing through the code using `ast.parse(code)` which is the function that reads the code/text.
2. Loop function that runs everytime step 3 occurs and then assigns the correct variable/parameter with `ast.Assign` function.
3. The criterion for naming the parameters is all uppercase letters (e.g. TRESHOLD) and check in the text for it. Return to step 2 if it’s correct with the function `t.id.isupper()`.
4. For the last step I’m using the `ast.literal_eval()` function which extracts the value attached to variable/parameter (e.g `(10, 0, 100)`) without running the code, which prevents crashes and errors.

Use of this can be found in the adaptability of the engine to whoever needs custom parameters to suit his desires for creating something unique.

4. Deployment and distribution

- PyInstaller is totally not needed for this program to work but is regardlessly added since the Tilemap studio is imagined to be easily transferable and in order to work flawlessly in any environment by merging all dependencies into a single standalone Windows Executable (.exe) file.

The core rendering Algorithm

This part of the Thesis paper explains how to achieve the desired effect described in the Introduction section step by step using simple python language and some smart mathematical methodes. The first section is dedicated to establishing the grid, then filling it with density, in the middle part we will focus on coloring as a stepping stone and then continue to solve the complex problem of edge detection. End is reserved for post processing.

1. Grid formation and downscaling

Before appling effects and complex mathematical functions we need to decide the size of our ASCII characters and how much space they are going to take in the image. We need to find a balance between readability of the ASCII characters and loss of detail. If we make our ASCII characters bigger, they will be more visible and in greater resolution but we will lose details on the image, this is the problem on the other side as well. If we choose ASCII characters that are in smaller resolution, we would have a more detailed image since there are smaller characters to define them, but they then would not be distinguishable as font/ASCII characters. In order to preserve as much detail as we can our characters need to be as small as possible but just big enough to be legible. For this project I decided to go with 8x8 pixels resolution characters since it is a power of 2 and it is fairly small with not much detail loss.

With that in mind, now every pixel (which are 1x1), should be replaced with the 8x8 ASCII character. This means that we need to modify our pixels to be the same size as the characters. Downscaling the image by destroying enough detail we can condense some N times M pixel size image into right dimensions. The algorithm "trims" the input image, discarding any excess pixels at the edges that do not fit into a complete 8x8 block. With this part finished we can treat the image not as a collection of pixels but as a matrix of 8x8 "cells".

```
tile_size = 8
    h, w, _ = src.shape
    # Integer division ensures dimensions are multiples of 8
    h_trim = (h // tile_size) * tile_size
    w_trim = (w // tile_size) * tile_size
    src_trim = src[:h_trim, :w_trim]
```

Original image vs downscaled image.

2. Luminance analysis and Quantization

With our image downscaled and trimed to the right value, we can establish a relation between pixels and ASCII characters that will dictate where what should go. The simplest and the best option is relating the luminescence/brightness values of the pixel to specific ASCII characters. We can get the luminance of the image by simply desaturating (removing color) the image. To achieve this effect, we must use something called the "human eye" formula for extracting correct luminance values (how much each pixel should be dark or light depends on its RGB value) and it looks like this: `lum = 0.299*r_ch + 0.587*g_ch + 0.114*b_ch` . If we simply averaged the colors (`(Red + Green + Blue) / 3`), a pure blue pixel (`0, 0, 255`), and a pure Green pixel (`0, 255, 0`) would technically have the exact same brightness value (85). If you look at the pure green color and pure blue color in real life, the green color would be blindingly bright and the blue would look dark and deep. This is because the human eye perceives brightness through "Cone" cells that are different regarding to which color they are dedicated.

- Green Cones (M-cones): We have the most of these, and they are most sensitive to light. So that is why we multiply it with the highest number (0.587). It contributes ~60% of perceived brightness.

- Red Cones (L-cones): We are moderately sensitive to red. It gets the medium multiplier of 0.299.
- Blue cones (S-cones): These are surprisingly insensitive to brightness and they mostly just help us distinguish color contrast. This is why blue is multiplied with 0.114. It contributes to only ~10% of perceived brightness.

We follow the biology of the human eye so the blue pixels get darker and green brighter for best match between desaturated and colored look.

```
# We must convert to float to do decimal math
img_float = small_fill.astype(float)

# Split channels (OpenCV uses BGR order)
b_ch, g_ch, r_ch = cv2.split(img_float)

# The "Perceptual Luminance" Formula
lum = 0.299*r_ch + 0.587*g_ch + 0.114*b_ch
```

Problem arises with this step as there are many possible variations of luminence values (0-255), which is way more than our ASCII characters. This is solvable by adding more ASCII characters, or the easier way by Quantizing the values to a larger N number of “batches” to have a smaller set of possible values. For this example I wanted 10 ASCII characters so the number of Quantization should be the same for best results. We are now rounding all possible values to one of 10 values.

```
# We normalize brightness to 0.0-1.0 (lum / 255.0)
# Then multiply by number of buckets (levels - 1)
# Then round down (floor) to get an index: 0, 1, 2... 9
bucket_index = np.floor((lum / 255.0) * (levels - 1))
# To "see" the buckets, we multiply the index back up to 0-255
# e.g. Index 0 -> 0 (Black), Index 9 -> 255 (White)
brightness_step = 255 // (levels - 1) quantized_view = (bucket_index *
brightness_step).astype(np.uint8)
```

Biologically correct desaturation and Quantized image

3. Texture mapping

With clear correlation between pixel's luminescence values and size and our predetermined ASCII character rules (8x8 size and 10 of total characters) we can replace the pixels. Before doing that we need Atlas texture to hold all of our ASCII characters and from which they should be sliced by our program and placed in the right position. Our ASCII characters need to communicate brightness values so a dark pixels are going to be replaced with e.g. `(space), ., -, ', *` while brighter pixels should be represented with: `$, %, #, &, @`.

The Atlas texture containing ASCII characters that I created using Adobe's Photoshop looks something like this.

Actual size Black pixel is represented by (space).

Now we use the luminance for the coordinate for our algorithm to place the correct character into its place. Slicing the Atlas texture is what comes first, cutting out 8x8 pixel squares and storing them in the list.

```
# 1. Convert Atlas image to a NumPy Array
atlas_np = np.array(atlas)
aw, ah = atlas.size
tile_h = ah // 2

# 2. Slice the Density Characters
for i in range(aw // tile_h):
    crop = atlas_np[0:tile_h, i*tile_h:(i+1)*tile_h]
    density_sprites.append(crop)
```

The logic is based around `bucket` which is our "Paint-by-Numbers" canvas. `density_lookup` is our palette (the list of 8x8 we created earlier). And with the function `density_lookup[bucket]` we tell NumPy to go and look at the number on the canvas and replace that number with the image at index 5 from the lookup table. This creates 4-dimensional Array `(rows, Cols, 8, 8, 4)` (A grid of Rows x Cols, where every cell contains an 8x8 pixel image with 4 color channels (RGBA)) which is why this is such a low performance cost methode for doing this kind of rendering.

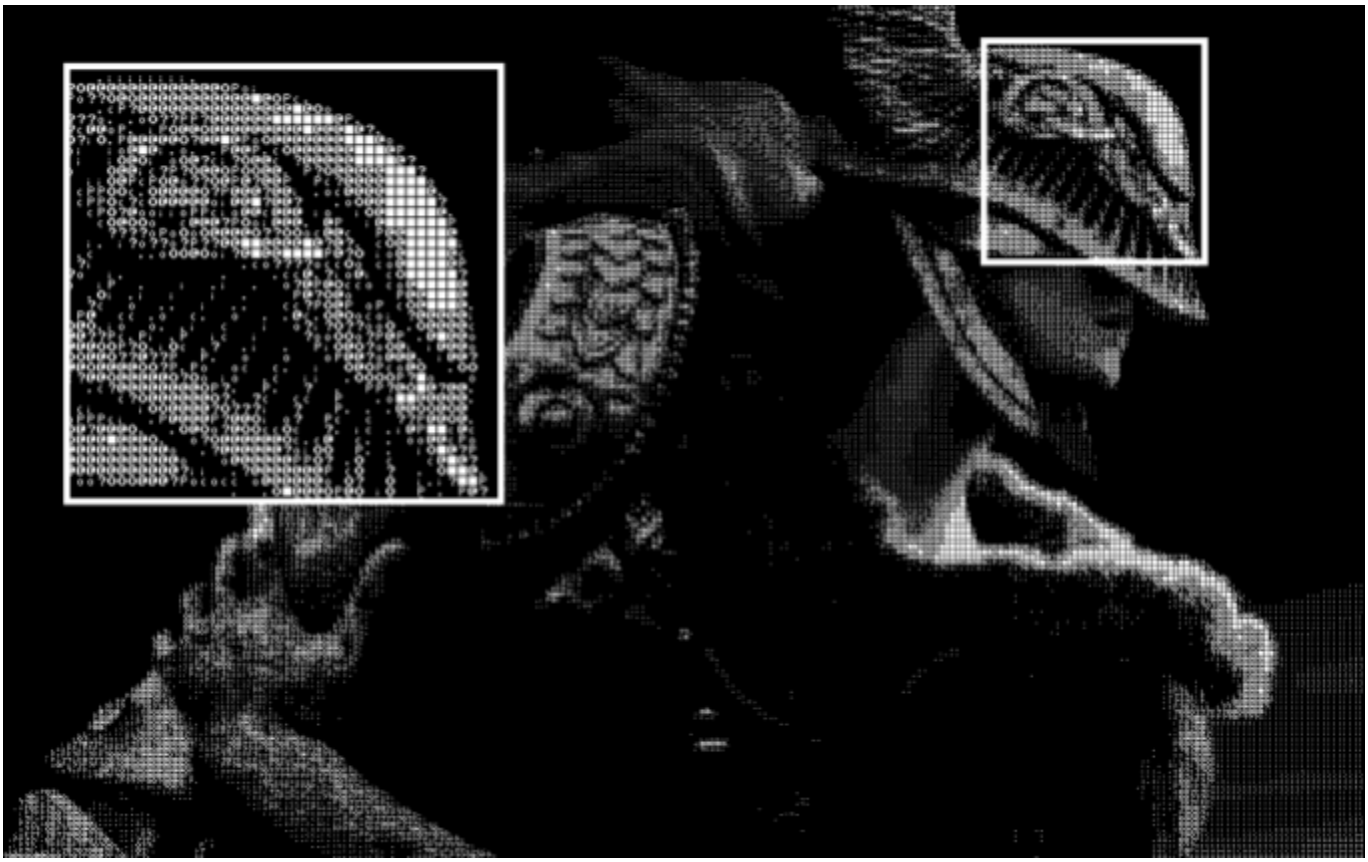
```
# 1. Convert list of sprites to a NumPy Array (The Lookup Table)
    density_lookup = np.array(density_sprites)

    # 2. THE REPLACEMENT (Vectorization)
    # Instantly replaces 0-9 indices with 8x8 pixel arrays
    final_grid = density_lookup[bucket].copy()
```

The final_grid is currently only a grid of tiles that looks like a checker board where every square is separate. To display it correctly on a monitor we need to make it look like a flat 2D image by rearranging the data so the pixels line up correctly. We move the “Tile Y” axis next to the “Grid Row” Axis, and merge them.

```
# Turns (Rows, Cols, 8, 8, 4) -> (ImageHeight, ImageWidth, 4)
final_image_rgba = final_grid.swapaxes(1, 2).reshape(h_trim, w_trim, 4)
```

The result is something that already looks like ASCII art!



4. Chromatic integration

Coloring our ASCII characters and adding color to the image is a fairly simple task. For best understanding of our coloring solution we must imagine that the ASCII characters are not “white text”. In math terms they are **multiplier masks.**

- Black part of the letters (0.0): Anything multiplied by 0 is 0 (black/invisible)
- White parts of the letters (1.0): Anything that is multiplied by 1 stays the same so it is visible.

We use this kind of mask to cut out the shape in which our colors are to be set.
By creating two different modes with slight variation in code we can achieve very different outcomes.

First mode is Static intensity mode or (Monochromatic) in which the system assumes the light source is uniform across the entire screen. The perceived brightness of a region is determined solely by the density of the character used.

The second mode is modulated Intensity or so called Dynamic brightness in which the brightness is not only derived from the density of the characters but also with its color. This methode can make more details stand out with finer lighting.

Over all of what we just said, by downscaling the original image again and mulitpliying that with our multiplier mask we can get the original colors or any color theme we want. Mathematical implementation would look something like this:

Let $\boldsymbol{C_{theme}}$ be the RBG color (e.g original downscaled image color data), $\boldsymbol{L_{source}}$ be the normalised luminance (0.0 to 1.0) of the original pixel, and $\boldsymbol{M_{ascii}}$ be the binary mask of the character shape (0 to 1).

- Static Intensity

$$Pixel_{final} = C_{theme} \times M_{ascii}$$

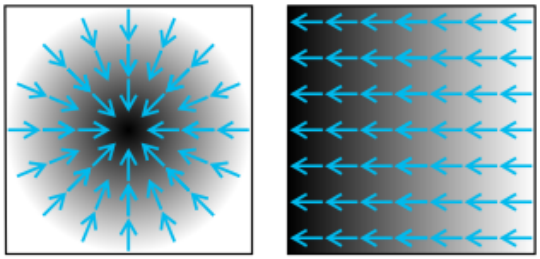
- Modulated Intensity (Dynamic brightness)

$$Pixel_{final} = C_{theme} \times L_{source} \times M_{ascii}$$

Dynamic brightness turned on

5. Edge detection

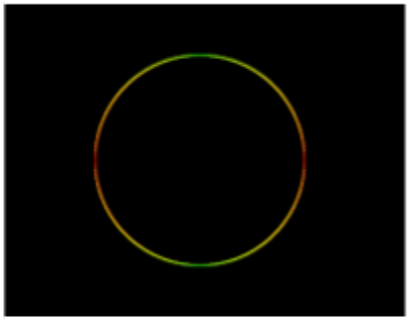
Simple edge detection algorithms cannot be the solution for our problem since we have to draw the edges in ASCII characters that conform to a curve edge. So we want to draw `_ , | , / , \` that follows objects contour and for that we also want to know the angle of the edge. For achieving our goal we will use **Sobel filter** because sobel isn't really an edge detection filtrer but **a gradient approximator**. In computer graphics the gradient is a vector field where each vector points in the direction of the greatest change.



Applying the Sobel filter to our image (for which we will use a simple black circle on white background for this) we get a distinct outline on the edges where our circle once was.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} *$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} *$$

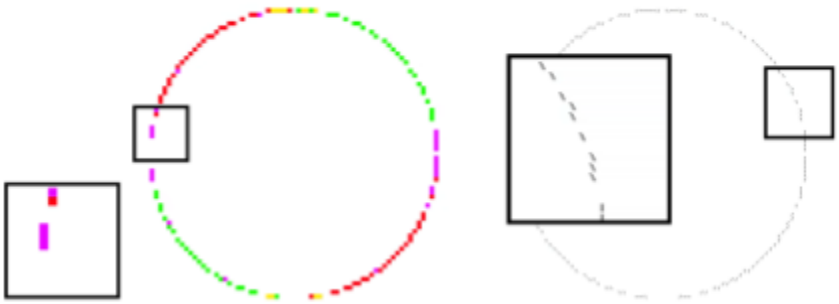


Sobel Kernel used to achieve effect Original image Result

Since these are vectors in 2-dimensional space we can extract their angle using function:

$$\theta = arctan(\frac{dy}{dx})$$

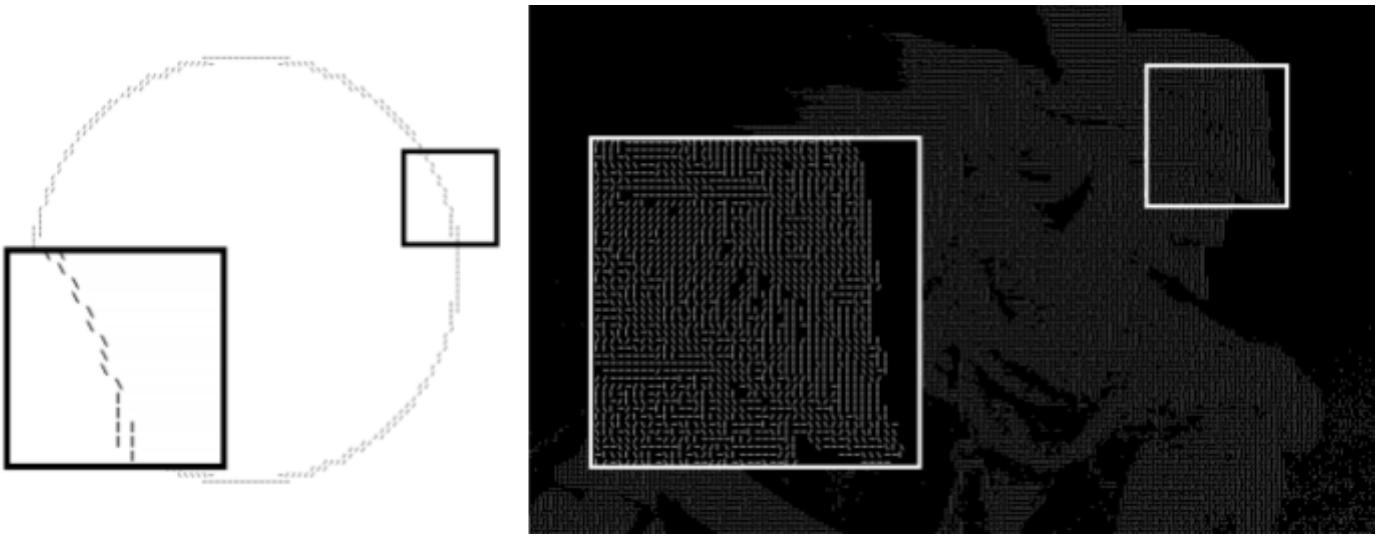
There being a lot more angles then our 4 ASCII characters we again need to quantize the values in 4 buckets, downscale the image and draw our ascii circle.



Quantized, but poorly conserved edges but the proof of concept is there.

In this format it is maybe hard to see but downscaling and quantizing the image poorly conserves edges and we are left with lots and lots of gaps as seen in the image above. To keep the lines as solid and consistent as we can before downscaling the image, determine if something should be an edge using Vectorized logic composition or so called “voting system”.

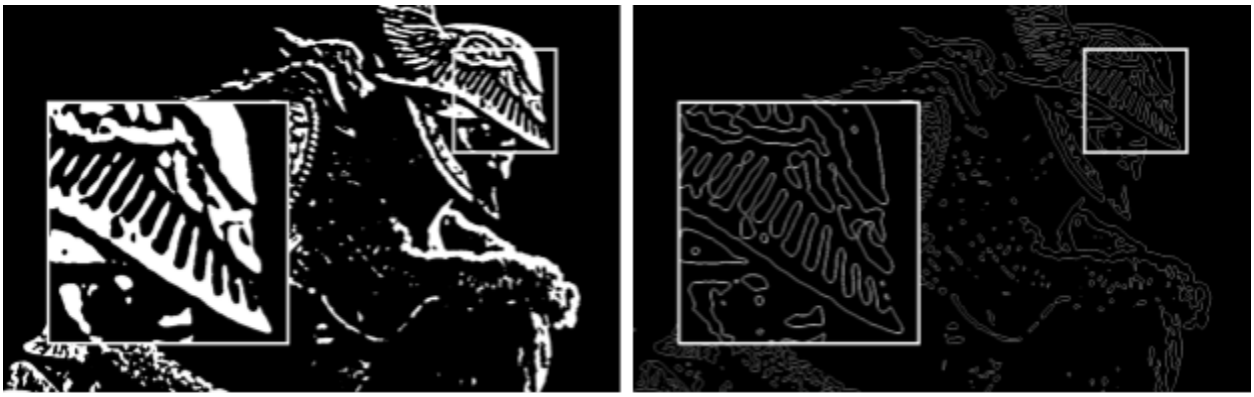
We have a single 8x8 tile in the downscaled version with 64 pixels underneath. Since we cannot draw 64 different lines, we have to pick one ASCII character that represents the majority opinion (pixel data) of that tile. There are 4 options and if the majority of pixels go in one of the categories (e.g. horizontal line), and just a couple go into different one, we draw that ASCII character that belongs to the majority’s category. Threshold is determined with lack of votes. So if the leading categorie wins by 3 votes only we declare that the election was invalid and the edge is not drawn. This gives us much more consistent lines when downscaling.



Introducing Sobel filter with more complex imagery leads to terrible results. Since the Sobel filter is not real edge detection but rather just a gradient approximation, it detects every little change in the image and declares it edge.

Sobel applied to image without downscaling

To achieve our goal of astheticaly pleasing edges we will add another filter right before computing Sobel. Difference of Guassians (DoG) is a band-pass filter used for isolating edges. It works by subtracting a “blurry” version of the image from a “slightly less blurry” version. After appling DoG filter and then thresholding the values (so that if high change is detected, it means it is an egde, otherwise it is not) we again run the Soble filter to extract angle data.



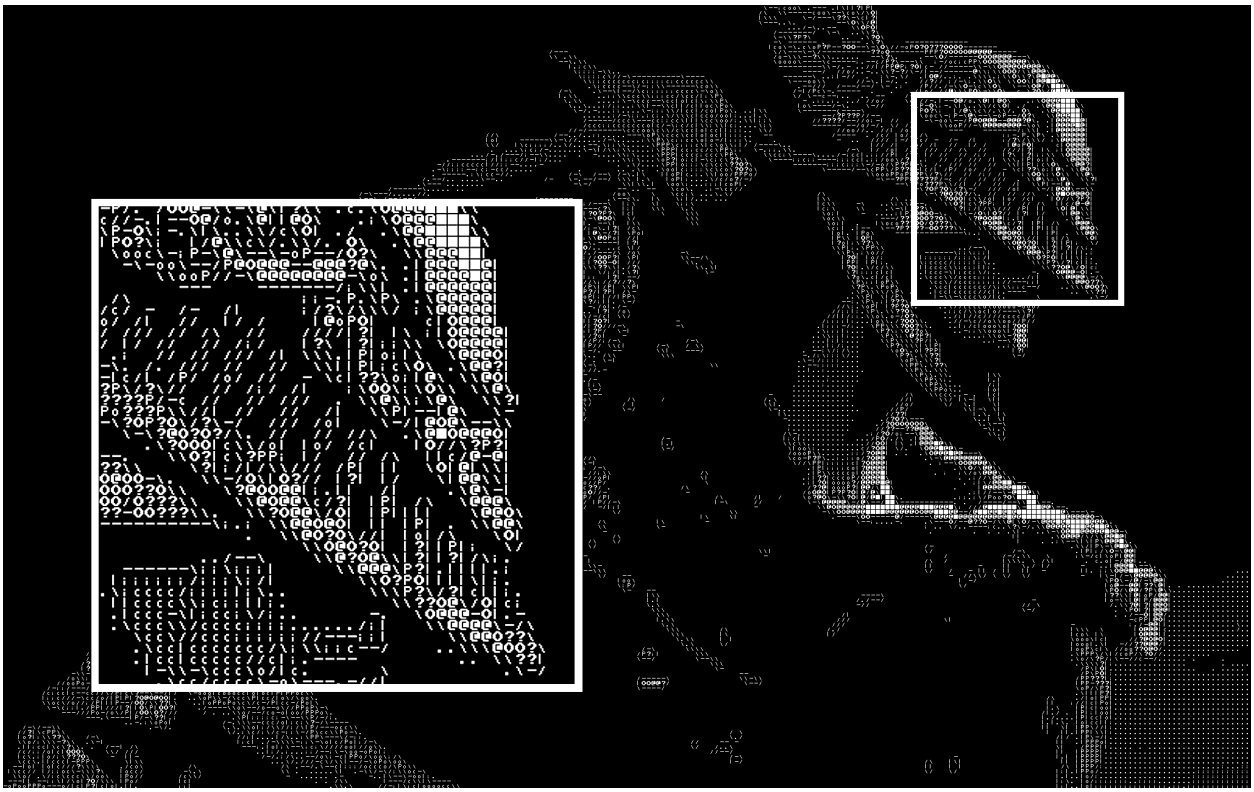
Difference of Gaussians and Sobel filter layed on top of it.

We are now effectively running a simplified **Layer Blend** operation, similar to one in Photoshop, with two new layers generated in parallel:

1. Bottom layer (fill of ASCII characters or Density)
2. Top layer (Edges)

Having something called **Alpha mask** here is very helpful since we need to draw the top layer on the bottom layer and know which tile of fill ASCII characters (Density) needs to be replaced.
When the edge is drawn, that tile of the alpha mask gets value 1, if the edge is not drawn, in that place value stays 0. Applying this mask onto the bottom layer will make us empty slots where an ASCII edge is to be drawn. The mathematical formula for **Apha blending** looks like this.

$$Final = Density \times (1 - Edge_{\alpha}) + Edge \times Edge_{\alpha}$$



6. Post and pre-processing

There are two steps that qualify as a post-processing in our algorithm that we didn't add still. The first one is not a real post-processing effect since it is done first in line with our algorithm and helps us determine how the image is going to look after our main ASCII algorithm. Those effects are simple **contrast** and overall **brightness** controls.

- Contrast expands the gap between light and dark parts of the image. This is one of the more important features since it helps the quantization catch wanted detail or reverse, not include detail.
- Brightness shifts the entire range up or down.

Math formula we are going to use is:

$$Pixel_{new} = (Pixel_{old} - 128) \times Contrast + 128 + Brightness$$

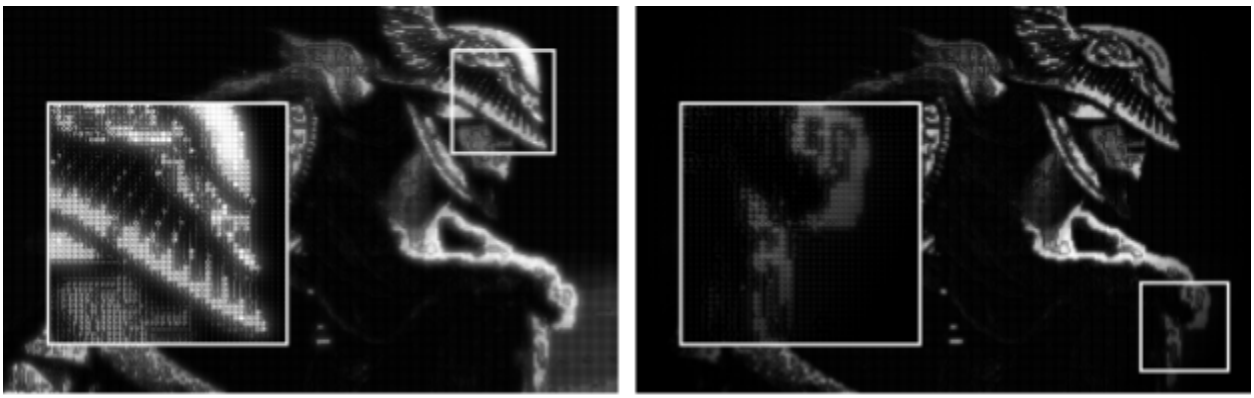
- We subtract 128 to move the pivot point to the mid gray (middle of the RGB values). This ensures that when we increase contrast, shadows get darker and highlights brighter simultaneously.

The second part of post-processing is one that qualifies as real and true post-processing since it is applying effects after the main pipeline is over. This part includes effects such as **bloom** and **vignetting**.

For the Bloom effect we want to simulate light scattering and pixels glowing outside of their borders. To achieve this effect we need to threshold the brightest parts of the image (e.g. ▉ symbols and @ symbols), blur them with heavy Guassian blur (just like the one we used to create DoG filter for our image) and then add them on top of the original image.

Vignetting is an important effect that is an unwanted reduction of an image's brightness or saturation toward the periphery compared to the image center caused by camera settings or lens limitations of primitive cameras. Nowadays it is a useful and often simulated effect that helps better frame the subject in the centre of the lens. To achieve the desired effect we can use the Euclidian distance of each pixel from the centre of the image to drive our vignetting math.

- Center: Distance = 0 (Multiplier 1.0 => No Change)
- Corners: Distance = Max (Multiplier 0.0 => Black)



Bloom and Vignetting effect added on top of the edited image with dynamic Brightness turned on.

Color themes are implemented by simply adding RGB (here in different format for convenience of a program we are using BGR system) to a python dictionary and assigning it a string name (e.g. Matrix).

```
# Define the dictionary (first BGR value is for the letters and the other is for background)
    schemes = {
        "Black & White": ((255, 255, 255), (0, 0, 0)),
        "Matrix":        ((0, 255, 50),    (0, 10, 0)),
        "Neon Demon":    ((255, 105, 180), (20, 5, 30)),
        "Firelink":      ((255, 140, 0),   (30, 10, 5)),
    }

    # Get user selection
    fg_color, bg_color = schemes.get(preset_name, schemes["Black & White"])
```

Chapter II: 3D-game implementation and ReShade

The last 10 pages were about pushing the 2D Tilemap Engine to its maximum. It is the best it can be done in 2D context, but since the motivation of the project from the start was implementing ASCII art into modern graphics systems wouldn't it be a shame if we don't try to implement our algorithm into games and improve it with additional info we can extract from 3D environment that games operate in. For this example I decided to use two games and one handy open-source post-processing injector softwear called ReShade.

Architecture of ReShade and algorithm implementation

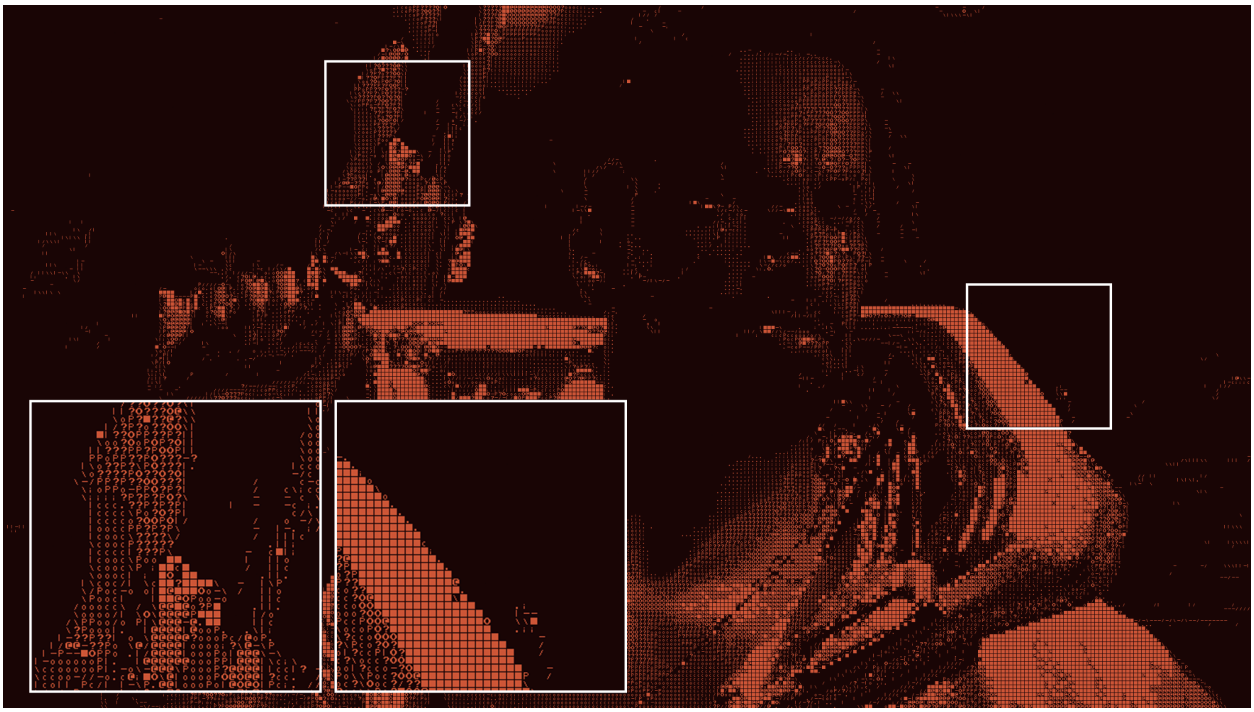
ReShade basically works like a "middle man" between your GPU and what comes on the screen, it intercepts the communication between the game engine and the GPU driver and in that interception we can alter the data sent to the **GPU** using simple but different programming language called HLSL (High-Level Shading Languge) and tell it how to render the game. The difference between our 2D Tilemap Engine which uses CPU and NumPy operations for low-performance costs and ReShade applying our algorithm directly to the GPU is in even lower performance costs which enables us to get results at some 70 to 80 frames per second when the game is natively running 90 frames per second (on my hardwear talking about Cyberpunk 2077).

An important factor in understanding how ReShade works is understanding that it operates entirely in **Screen Space**. It doesn't have access to the game's 3D object data (vertices, polygons or light sources). Instead, it has to operate on two specific 2D buffers that the game engine has already rendered and is providing.

1. The color buffer (`ReShade::BackBuffer`) is the final RGB image that is intended for projection, which we use for the algorith we made and explained earlier.
2. The depth buffer (`ReShade::DepthBuffer`) is a grayscale map where every pixel represents a distance (Z-value) from camera to objects on the screen, which will give us some room to improve our algorithm in edge detection.

Entire Architecture of ReShade is based on Screen-space post-proccessing effects or simply called (SSPE). The ASCII conversion must occur solely by analyzing these two 2D textures. Game on which we will see the final results of our algorithm will be Cyberpunk 2077.

Just applying our algorithm in ReShade we can get pretty interesting results.



From this screenshot we can notice how our algorithm returns the image (Due to limitations of this paper form I cannot show real-time video output, recorded screen in a video format will be uploaded to YouTube platform and their respected links can be found down below in the gallery section). Looking closely onto our output we can notice how luminance determination, texture mapping, color themes (I picked a custom option and made this low-contrast monochromatic mode to ease eye fatigue) work flawlessly. Edge detection is good in some parts of the image and in others it can lack behind due to a number of reasons. High resolution rough textures tend to be oversaturated with edges drawn by DoG and Sobel engine; if the background and foreground meet with the same color, it cannot detect edges; If the character is standing in the shadow and the lighting is poor there is no way for the engine to detect any edges. These are just some of the problems we can come across not only with ReShade in games but also in 2D images and videos that do not provide the already mentioned Depth Buffer which is now going to be our way to fix these edges, and add some more flexibility to our algorithm.

Implementing Depth buffer in our algorithm

In a standard 3D rendering pipeline, the Z-buffer is non linear, it offers a high range of data from objects close and distant from the camera. To perform accurate egde detection we first need to linearize the data. The implementation is using the ReShade helper function based on the math equation:

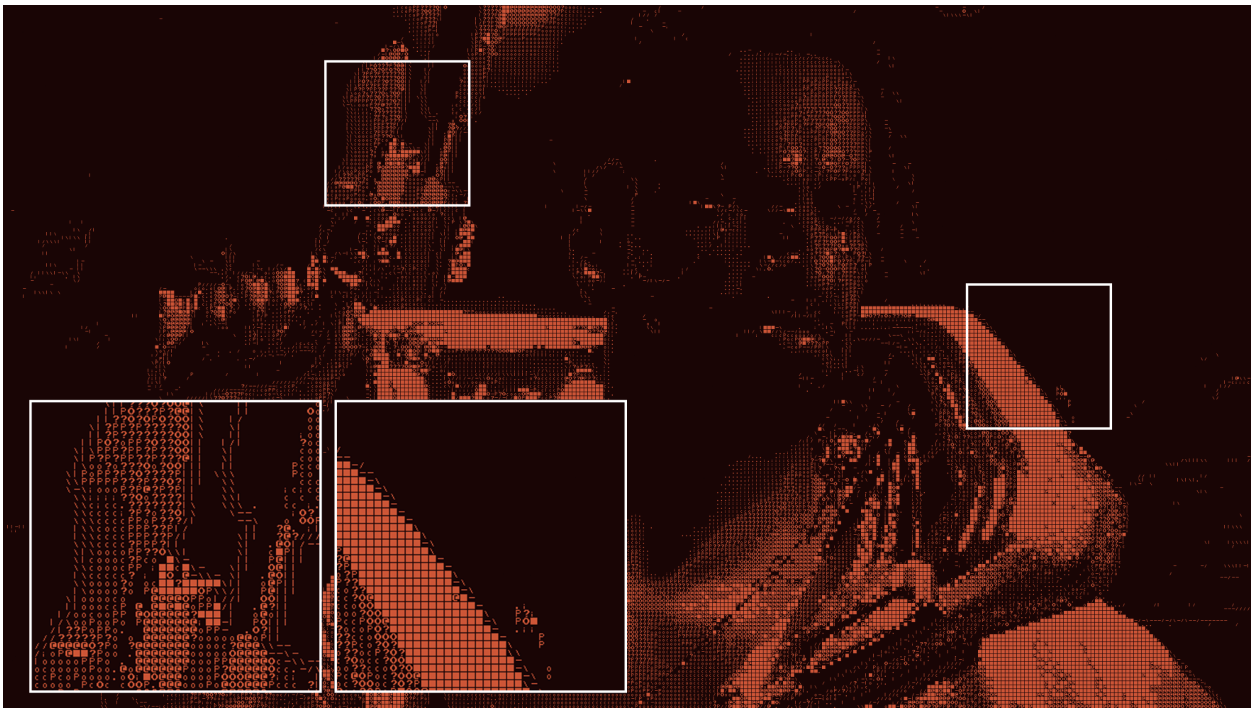
$$d_{linear} = \frac{2.0 \cdot z_{near} \cdot z_{far}}{z_{far} + z_{near} - z_{buffer} \cdot (z_{far} - z_{near})}$$

In the HLSL language we can run this function by simply typing command like

```
float depth = ReShade::GetLinearizedDepth(gridUV);
```

This returns a normalized float value between 0.0 (camera lens - black) and 1.0 (horizon - white). This is the fondation for the atmospheric fog and background separation logic.

With depth data present in this form, we can implement a couple of interesting options that should give more flexibility to work with.



Looking at this new image we can clearly see the edge fidelity being increased by a lot. Also the edges appear thicker with a lot more consistency. This is achieved by combining two methodes together to get the best of both worlds. We can also use the depth buffer to impact how the background and foreground of our image behave and look. For example we can bring the brightness of the background down and make our foreground object pop out more by creating much less visual noise in the process.

Effect of lowering the density of the luminance map and making individual ASCII characters darker based on the depth buffer

Finally this goes to show that the most efficient way to stylize a 3D environment is not to treat it as a 2D image, but to extract as much available data from its 3D nature. By moving the edge detection logic from the color buffer to the depth buffer, we resolved the fundamental conflict between texture noise and geometric outline that is the issue for any traditional edge filters.

Gallery

Some of the artworks I was able to create were printed out and included into the paper version of this thesis paper but the large majority of the images and video examples are uploaded to YouTube and other image-viewing websites. You can scan these QR codes to see the full gallery and video example uploaded to my YouTube channel.

Photo gallery link

Github and open-source repository

Conclusion