CT029-3-2-ISE

IMAGING & SPECIAL EFFECTS

APU2F2402CS(AI)

IN–COURSE ASSESSMENT
(GROUP PROJECT)

PREPARED BY: GROUP 15

TP Number	Student Name
ANGELINA LEANORE	TP072929
BRITTNEY LAU KO XUAN	TP065461
CHANG ZHENG FANG	TP066899
CHEAH MEI SUEN	TP064568
CHUA JUN YI	TP065882

Table of Contents

Introduction

This assignment is about the exploration of digital image processing. We took some ordinary Tokyo cityscape images to reimagine them into different scenes such as futuristic, apocalypse, post-apocalypse and more. The goal is to apply various special effects using Python libraries to create compelling visuals. We have utilized different techniques such as color grading and tinting, adding atmospheric effects, overlaying textures, adding objects and elements, light and shadow manipulation, creating composite Images and storytelling elements. By combining creativity and technical knowledge, the project will result in creation of unique digital images that can blend the real world with futuristic or disaster-stricken elements.

Story Overview

The story starts with Japan inventing superconductors, accelerating the advanced of technology of the whole country especially Tokyo. Thus, the city of Tokyo has become very modern and advanced. One day, the governing party's leader was assassinated, and liberals took over the city. They practiced corruption and civilians' quality of life quickly declined. The weakened defenses and infrastructure created an opening for opposing countries to invade the city of Tokyo, by sending armies and bombing the city. The city of Tokyo drowned in flames and toxic smoke from the bombs. The city of Tokyo became a wasteland, where buildings were abandoned, and weeds grew all over. Until one day, the toxic fumes finally settled down, and the city started to rebuild.

Major techniques used

Imaging and special effects techniques will be used to create the scenes from our storyline, giving a visual input and perception to provide more context towards the story, and convey the feelings and details for better storytelling. Our goal is to create some futuristic scenes, and post-apocalyptic scenes.

The team has applied major techniques using the Pillow and OpenCV library from python. Imaging and special effects techniques included the use of color grading to set the tone of our scene. Blue and purple tones were used to form futuristic tones, whereas red, orange or washed-out colors were used to convey the barren post-apocalyptic city. The team has also relied on overlaying textures and adding objects onto the pictures to create more detailed and complex images. Elements like neon lighting and cars were used to produce futuristic scenes, and dust, cracks, weed, and debris were used to add a post-apocalyptic touch to the city images. Finally, the team also utilized a lot of storytelling elements and created composite images to tailor the images to the storyline, blending multiple scenes, creating reflections, and introducing population hints. Other methods such as HJDR effect, adding atmospheric effects were also used to portray weather, smoke, fires, and more.

Scene 1: Before apocalypse (Futuristic)

1.1 Storyline and Images

Figure 2.1.1 Futuristic Tokyo Street with Flying Car image

Tokyo had become a beacon of human innovation, a city where technology and imagination intertwined seamlessly. Years of breakthroughs in superconductors and sustainable energy had transformed the skyline into a mesmerizing display of glowing skyscraper filled the day with colour and wonder.

Above the bustling streets, flying cars glided silently, zipping across the cityscape. Every corner of the city seemed alive, bathed in the soft, otherworldly glow of advanced tech woven into everyday life. It was a city where dreams met reality, where the once impossible had become an integral part of daily existence. The people of Tokyo lived in a constant dance with technology, surrounded by shimmering lights that illuminated their path toward an ever-brighter tomorrow.

Figure 2.1.2 Animated Flying car in Tokyo City GIF

In the heart of a futuristic Tokyo, the skyline glittered with unprecedented brilliance. The iconic Tokyo Tower stood at the city's core, bathed in a golden glow that mirrored the city's innovative spirit. Surrounding it, the towering skyscrapers that bathed in vibrant neon lights stretched towards the sky and creating a breathtaking urban jungle of technology and progress.

Above the city, the air was alive with the hum of sleek flying cars zipping through the purple-tinted sky. These flying cars gliding effortlessly between the towering buildings. The futuristic vehicles had replaced traditional roads and their presence a testament to the city's groundbreaking advancements in transportation. The city was a utopia of technology, where every facet of life was enhanced by the latest innovations.

1.2 Technique used

Technique	Method use	Sample Image
Original Image		

Color Grading and Tinting	• Conversion to HSV (Hue, Saturation, Value) color space
	• Increase saturation
	  - To intensify the colours and make them more vivid and neon-like
	• Adjust brightness
	• Shift Hue
	  - Favour cooler tones like purple, blue, and pink to emphasize a "neon" vibe
		

Adding Atmospheric Effects	• Random Glow Generation • Gaussian Blur - Create glow layer to create a smooth, diffuse glow effect • Blending - Make the neon spots blend naturally with the city image to give the scene a futuristic feel	
Adding objects and elements	• Reading and resizing flying car image • Handling Alpha Channel - blend the car's colours into the city background to make it semi-transparent. • HSV Adjustments on the Flying Car - Slightly reduce the brightness and saturation and the image is blurred to match the visual style. • Positioning and Blending - The flying car is placed at a specific position within the city image and the car's colour is blended with the city image using the alpha channel to maintain transparency.	

Light and Shadow Manipulation	• Glow Creation - A glow effect is created by drawing a circle around the flying car, simulating a trail. This circle is filled with a grey neon colour and then blurred using Gaussian Blur to diffuse the glow. • Blending the Glow - The glow effect is then blended into the city image with an intensity factor to control how strong the glow should appear.	

Technique	Method Used	Sample Image
Original Image	/	
Color Grading and Tinting	• HSV Color Space Manipulation - The image is converted to the HSV (Hue, Saturation, Value) color space, where the saturation is increased to make the colors more vivid, and the hue is slightly shifted to emphasize neon colours • Brightness Boost	

Adding Atmospheric Effects	• Neon Glow Spots - Simulate futuristic lighting in the city • Gaussian Blur - Applied to the glow spots, making them appear like soft and diffused light	
Adding Objects and Elements	• Flying Cars Overlay - The flying car images are adjusted and then placed on the city image.	

Light and Shadow Manipulation	• Light Trail effect - Glow effect is applied on the flying cars to simulate light trails, mimicking how light interacts with the surroundings as the cars move. This gives the impression of dynamic lighting in the scene.	
Creating Composite Images, Storytelling Elements	• Blending the Flying Cars - The flying cars are blended with the city scene using alpha blending. This creates a composite image by combining the car images with the background. • Animated Flying Cars - The flying cars move across the scene from one frame to the next, simulating motion. The scene changes over time, adding a narrative dimension to the composition.	

1.3 Screenshots of solution

Animated flying cars in Tokyo City

Figure 2.3.1.1 Flying Car image

```
flying_car = cv2.imread('flyingcar5.png', cv2.IMREAD_UNCHANGED)

if flying_car is None:
    raise FileNotFoundError("Flying car image not found.")

# Function to process and adjust the flying car image
def process_flying_car(flying_car, new_width, flip=False):
    car_height = int(flying_car.shape[0] * new_width / flying_car.shape[1])
    flying_car_resized = cv2.resize(flying_car, (new_width, car_height))

    # Check if the flying car image has an alpha channel
    if flying_car_resized.shape[2] == 4:  # RGBA format
        car_color = flying_car_resized[:, :, :3]  # RGB channels
        car_alpha = flying_car_resized[:, :, 3] / 255.0  # Normalize the alpha channel
    else:
        car_color = flying_car_resized
        car_alpha = np.ones_like(car_color[:, :, 0], dtype=np.float32)

    # Adjust the car's color to match the neon scene
    car_hsv = cv2.cvtColor(car_color, cv2.COLOR_RGB2HSV)
    car_hsv[:, :, 2] = cv2.normalize(car_hsv[:, :, 2], None, 0, 255, cv2.NORM_MINMAX)
    car_hsv[:, :, 2] = cv2.add(car_hsv[:, :, 2], -50)  # Reduce contrast by decreasing brightness
    car_hsv[:, :, 1] = cv2.normalize(car_hsv[:, :, 1], None, 0, 100, cv2.NORM_MINMAX)
    car_color_adjusted = cv2.cvtColor(car_hsv, cv2.COLOR_HSV2RGB)
    car_color_adjusted = cv2.GaussianBlur(car_color_adjusted, (5, 5), 0)

    # Flip the car image
    if flip:
        car_color_adjusted = cv2.flip(car_color_adjusted, 1)  # Flip horizontally
        car_alpha = cv2.flip(car_alpha, 1)  # Flip the alpha channel as well

    return car_color_adjusted, car_alpha, car_height
```

Figure 2.3.1.2 Code snippets to process the Flying Car image

Figure 2.3.1.2 is the code snippets to process flying car image to fit in the image for animation. After the image of flying car in figure 2.3.1.1 is loaded, process_flying_car function is defined to handle the resizing, flipping, and colour adjustments of the flying car image. The car image is resized to a new width and the height is calculated proportionally to maintain the aspect ratio. After resizing, the function checks if the car image has an alpha channel in RGBA format. If it does, the RGB colour and alpha channels are separated, and the alpha channel is normalized to a range of 0 to 1. The function

then converts the car's colour to the HSV colour space, where the brightness and saturation are adjusted to create a more muted and neon-like appearance. A slight Gaussian blur is also applied to smooth out the car's appearance. One of car is flipped horizontally which is the second car as it will fly to another way.

```
# 4. Adjust each flying car size
car_width1 = 400  # Width for the first car
car_width2 = 200  # Width for the second car (different size)
car_width3 = 250  # Width for the third car (different size)

# Flying car 1
car_x1, car_y1 = 700, 70
car_color1, car_alpha1, car_height1 = process_flying_car(flying_car, car_width1)

# Flying car 2
car_x2, car_y2 = 800, 500
car_color2, car_alpha2, car_height2 = process_flying_car(flying_car, car_width2, flip=True)

# Flying car 3
car_x3, car_y3 = 1400, 300
car_color3, car_alpha3, car_height3 = process_flying_car(flying_car, car_width3)
```

Figure 2.3.1.3 Code snippets of Setting Car Sizes and Positions

For the animation, three flying cars are created. Each of them has different sizes and starting positions. Car 1 is given a width of 400 pixels, Car 2 a width of 200 pixels which is smaller, and Car 3 a width of 250 pixels. The positions for each car are defined with (car_x, car_y) coordinates. The process_flying_car function is called for each car, where the respective width is passed in, and the resulting car images are processed including resizing, colour adjustment, and flipping in figure 2.3.1.2. This step ensures that each car has the correct visual appearance and is placed in the correct location within the scene.

```python
# Function to blend the flying car with the city image
def blend_flying_car(city_image_rgb, car_color, car_alpha, car_x, car_y, car_width, car_height):
    y1, y2 = max(0, car_y), min(city_image_rgb.shape[0], car_y + car_height)
    x1, x2 = max(0, car_x), min(city_image_rgb.shape[1], car_x + car_width)

    # Handle the case where the target region might be slightly smaller than the car
    if (y2 - y1) != car_color.shape[0] or (x2 - x1) != car_color.shape[1]:
        car_color_resized = cv2.resize(car_color, (x2 - x1, y2 - y1))  # Resize the car to match the region
        car_alpha_resized = cv2.resize(car_alpha, (x2 - x1, y2 - y1))  # Resize the alpha channel as well
    else:
        car_color_resized = car_color
        car_alpha_resized = car_alpha

    # Blend the flying car with the city image
    for c in range(0, 3):
        city_image_rgb[y1:y2, x1:x2, c] = (1 - car_alpha_resized) * city_image_rgb[y1:y2, x1:x2, c] + car_alpha_resized * car_color_resized[:, :, c]
    return city_image_rgb

# Function to add glow effect on each flying car (neon light trail)
def add_glow(city_image, car_x, car_y, car_width, car_height, glow_radius=30, glow_alpha=0.7):
    glow = np.zeros_like(city_image, dtype=np.uint8)
    glow_color = (169, 169, 169)  # Glow color (greyish neon glow)

    # Add a glow trail
    cv2.circle(glow, (car_x + car_width // 2, car_y + car_height // 2), glow_radius, glow_color, -1)
    glow = cv2.GaussianBlur(glow, (glow_radius * 2 + 1, glow_radius * 2 + 1), 0)

    # Blend the glow with the city image
    city_image = cv2.addWeighted(city_image, 1.0, glow, glow_alpha, 0)
    return city_image
```

Figure 2.3.1.4 Code snippets of Blending flying car and Adding Glow Effect

Figure 2.3.1.4 is the code snippets of blending flying car in the Tokyo city image and adding glow effect on the flying car. Once the flying cars are prepared, they need to be placed into the city scene. The function blend_flying_car is responsible for this task. It takes the city image and the car's processed image and blends the car into the city at the specified coordinates (car_x, car_y). The function calculates the bounding box of the car's position in the city image and checks if the car fits within that region. If the car's size does not match the region, the car is resized to fit. Alpha blending is then applied to merge the car with the background image, allowing for transparency effects where the car seamlessly blends into the scene. This process is repeated for each of the three cars, ensuring they are correctly placed and visible within the scene.

To enhance the futuristic and neon aesthetic, a glow effect is applied for each flying car using the function add_glow. This function creates a circular glow effect with a predefined radius on each car. The glow is drawn as a filled circle on a blank image and then blurred using a Gaussian filter to give it a smooth and glowing look. The glow is then blended with the city image using the cv2.addWeighted function, which combines the glow effect with the city background while controlling the transparency of the glow through an alpha blending factor.

```python
# Create the animated GIF
frames = []

for i in range(30):  # Create 30 frames of animation
    # Move the cars:
    car_x1 -= 20  # First car moves left
    car_y1 += 20   # First car moves down (diagonal movement)
    car_x2 += 30  # Second car moves right
    car_x3 -= 40  # Third car moves left

    # Create a copy of the base city image
    city_image_frame = city_image_rgb.copy()

    # Blend the cars into the scene
    city_image_frame = blend_flying_car(city_image_frame, car_color1, car_alpha1, car_x1, car_y1, car_width1, car_height1)
    city_image_frame = blend_flying_car(city_image_frame, car_color2, car_alpha2, car_x2, car_y2, car_width2, car_height2)
    city_image_frame = blend_flying_car(city_image_frame, car_color3, car_alpha3, car_x3, car_y3, car_width3, car_height3)

    # Add glow for each flying car
    city_image_frame = add_glow(city_image_frame, car_x1, car_y1, car_width1, car_height1)
    city_image_frame = add_glow(city_image_frame, car_x2, car_y2, car_width2, car_height2)
    city_image_frame = add_glow(city_image_frame, car_x3, car_y3, car_width3, car_height3)

    # Enhance yellow tones to make the tower light brighter
    city_image_frame = enhance_yellow_tones(city_image_frame, hue_range=(25, 45), saturation_boost=1.8, brightness_boost=1.5)

    # Convert to PIL Image and append to frames
    frames.append(Image.fromarray(city_image_frame))

# Save the frames as a GIF
frames[0].save('tokyocity.gif', save_all=True, append_images=frames[1:], duration=50, loop=0)
```

Figure 2.3.1.5 Code snippets of Creating Animated GIF

Figure 2.3.1.6 GIF of Animated Flying Car in Tokyo city

With the city and cars properly prepared, the animation is generated. The code creates 30 frames with one per loop iteration, with each frame representing a slightly different position of the cars to simulate movement. The positions of the cars are updated on each iteration such as Car 1 moves diagonally down and to the left, Car 2 moves to the right, and Car 3 moves left. For each frame, a copy of the city image is made, and the three cars are blended into the scene using the blend_flying_car function. The glow effect is also added for each car, and the yellow tones in the image are enhanced again to maintain consistency in the visual theme. After processing the frame, it is converted to a PIL image and stored in a list. Finally, all frames are saved as an animated GIF using frames[0].save, specifying the frame duration and loop behaviour. The resulting GIF has the cars moving across the city with glowing trails, creating an animated scene as shown in figure 2.3.1.6.

Neon-like effect Tokyo Street

Figure 2.3.2.1 Original Tokyo Street image

```
# Load the main city image
input_image = cv2.imread('Tokyo3.jpg')

# Convert the city image to RGB for proper display in matplotlib
city_image_rgb = cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB)

# 1. Enhance saturation and apply neon color effect (adjust color vibrancy)
hsv = cv2.cvtColor(city_image_rgb, cv2.COLOR_RGB2HSV)

# Boost the saturation to make the colors more vivid
hsv[:, :, 1] = hsv[:, :, 1] * 1.9  # Increase saturation
hsv[:, :, 1] = np.clip(hsv[:, :, 1], 0, 255)  # Clip values to valid range

# Boost brightness for a neon effect
hsv[:, :, 2] = hsv[:, :, 2] * 1.0  # Increase brightness
hsv[:, :, 2] = np.clip(hsv[:, :, 2], 0, 255)  # Clip values to valid range

# Shift the hue slightly to accentuate neon colors like purple, blue, and pink
hsv[:, :, 0] = (hsv[:, :, 0] + 20) % 180  # Shift hue towards more purple and blue

# Convert back to RGB
city_image_rgb = cv2.cvtColor(hsv, cv2.COLOR_HSV2RGB)
```

Figure 2.3.2.2 Code Snippets of Enhancing the Image with Neon Colours

Figure 2.3.2.3 Enhanced Tokyo Street image

Figure 2.3.2.1 is the code snippets to enhance the Tokyo Street image to make it more futuristic with neon colours. The saturation is increased by a factor of 1.9 to make the colours more vibrant while the brightness is slightly boosted although the multiplier is 1.0, so the effect is minimal. The hue is then shifted by 20 degrees to move the colour palette towards more vibrant neon colours like purple, blue, and pink. After these adjustments, the image is converted back to RGB format. The Tokyo Street image will then be enhanced from figure 2.3.2.1 to figure 2.3.2.3.

```
# 2. Add glowing spots to simulate futuristic lights with neon colors
glow_layer = np.zeros_like(city_image_rgb, dtype=np.uint8)

def add_neon_glow(layer, num_spots=10, max_radius=50, max_intensity=255):
    height, width, _ = layer.shape
    neon_colors = [(255, 0, 255), (0, 255, 255), (0, 255, 0), (255, 0, 0)]  # List of neon colors
    for _ in range(num_spots):
        # Random position and radius for the glowing spot
        center_x = np.random.randint(0, width)
        center_y = np.random.randint(0, height)
        radius = np.random.randint(10, max_radius)
        intensity = np.random.randint(100, max_intensity)  # Random intensity of glow

        # Choose a random neon color for the glow from the list
        neon_color = neon_colors[np.random.randint(len(neon_colors))]

        # Draw a circle on the glow layer (we use a bright color for the glow)
        cv2.circle(layer, (center_x, center_y), radius, neon_color, -1)

    return layer

glow_layer = add_neon_glow(glow_layer)
glow_layer_blurred = cv2.GaussianBlur(glow_layer, (15, 15), 0)
city_image_rgb = cv2.addWeighted(city_image_rgb, 1.0, glow_layer_blurred, 0.5, 0)
```

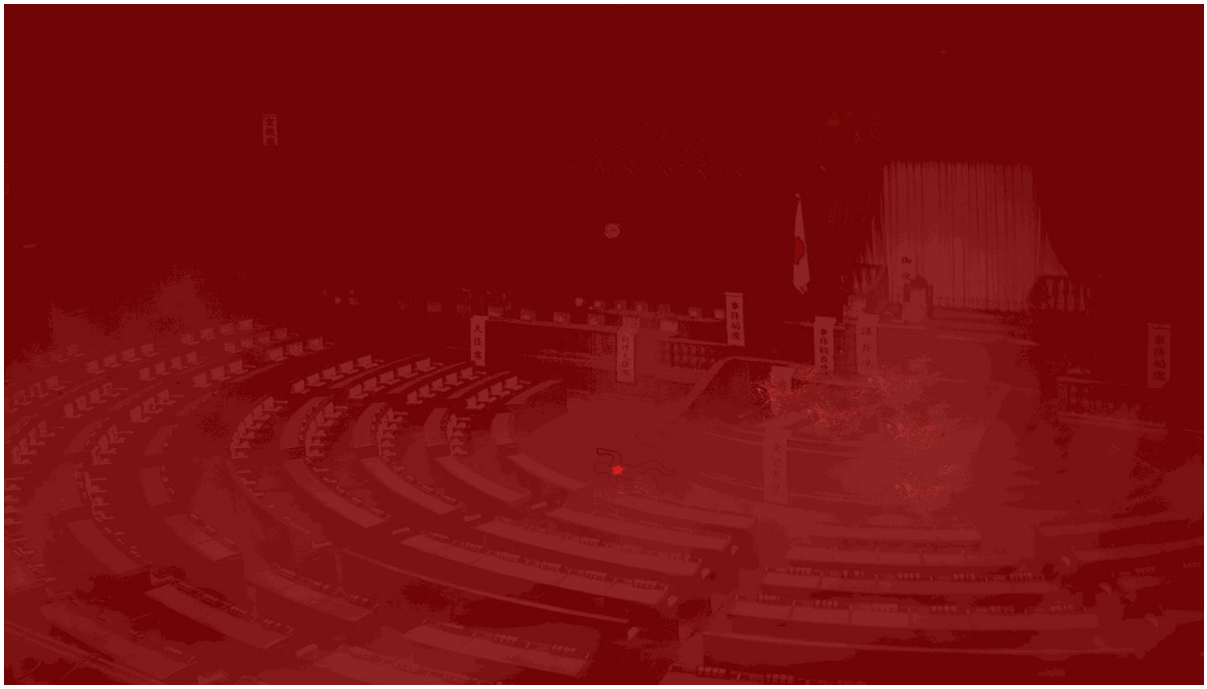
Figure 2.3.2.4 Code snippets to Add Glowing Spots

Figure 2.3.2.5 Enhanced Tokyo Street with Neon lights image

To simulate futuristic and glowing neon lights in the city scene, a glow layer is created. This layer is initially black (all zeros) and then populated with glowing spots. The function add_neon_glow adds a specified number of glowing spots that will be randomly placed. Each spot has a random position, random radius and random neon colour on the canvas. The random glow intensity is set between 100 and the maximum possible intensity which is 255 and the random neon colour chosen from a predefined list such as pink, cyan, green, and red. The glowing spots are drawn using cv2.circle() and then blurred using Gaussian blur to create a smooth glow effect. The glow layer is then blended with the city image using cv2.addWeighted(), which combines the city image and the glow effect with a specified weight which result in figure 2.3.2.5.

Scene 2: Fall of Government

1.4 Storyline and Images



Despite the advancements in technology and its innovations elevating the city of Tokyo into a global beacon of innovation, ideologies keep the city deeply divided, in a sudden turn of events, within the gran halls of the National Diet Building, the assassination of the governing party's leader and the Prime Minister of the nation. THE PRIME MINISTER IS DEAD!!!, shouts a lawmaker evacuating the scene, and the parliament, which was once a symbol of unity and progress, is now ground zero for a bloody revolution and Coup d'état. The futuristic advancements that bridged gaps between Tokyo's people fail to mend the ever-widening divide in beliefs. Smoke filled room with alarms blaring, blood stains the polished floors as the new regime seizes power, sowing seeds of corruption that threaten to dismantle the city's fragile unity and everything the city stood for.

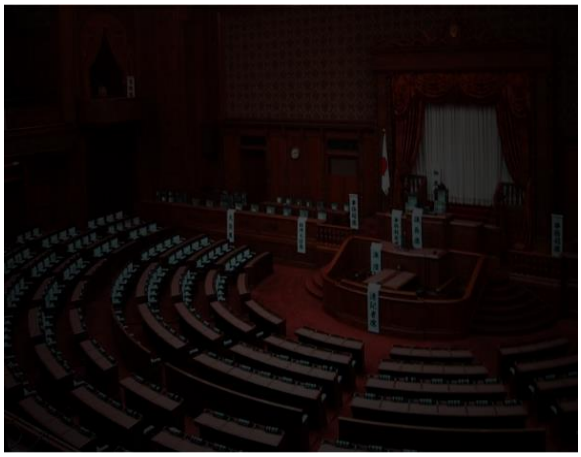
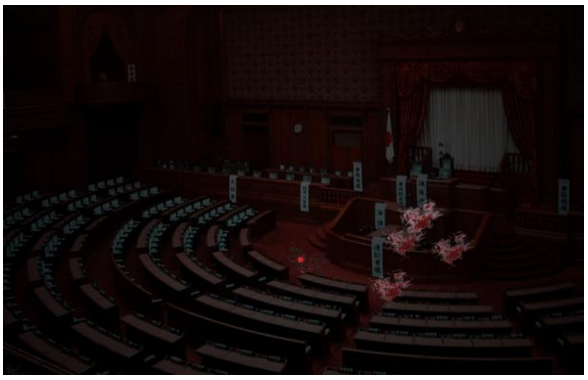
With the new regime in power, the citizens of Tokyo once proud of the cities technological marvels, now find themselves betrayed by those in power, corruption becomes rampant and the quality of life in the city plummets, the citizens let their despair and anger boil into action, Crowds gather before the National Diet Building, their silhouettes illuminated by the glow of protest banners, each marked with red—a symbol of both defiance and blood spilled in the name of justice. Chants rise into the smoke-tainted sky, an unrelenting demand for accountability. Despite the technological advancements surrounding them, the masses wield only their voices and determination against a government that has failed them. The once-pristine facade of the Diet Building looms in the background, shrouded in a haze of violent unrest and decay—a silent witness to the turmoil of its people.

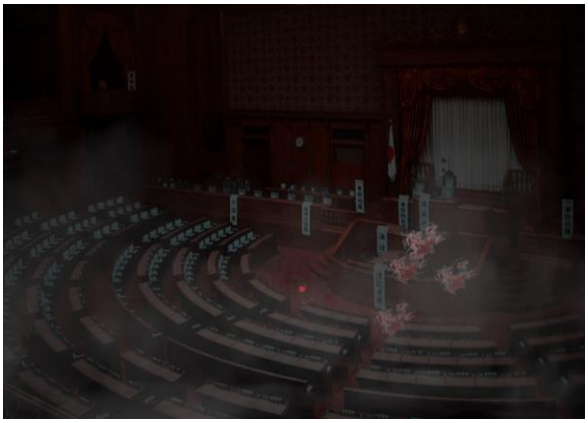
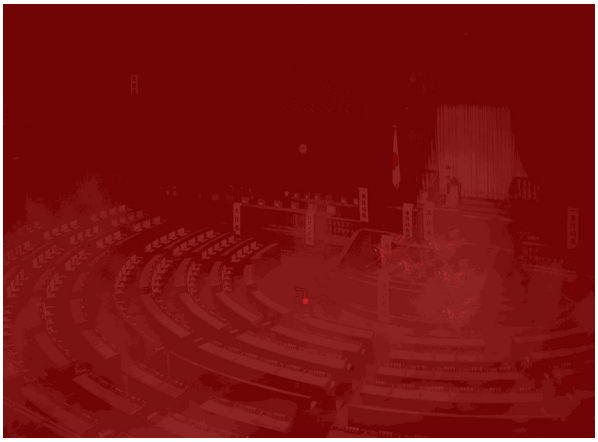
As the rebellion gains momentum, the armed forces of Japan (JSDF) was divided into multiple factions, some supporting the corrupt regime while others were sympathizers of the rebellion, they both saw the royal family as an important asset in legitimizing their rule of the nation, the old symbol of Japan, the Imperial Palace, becomes a central battleground, once a symbol of cultural heritage and unity of the people of Japan, the palace now stands under siege, a military helicopter hovers overhead, deploying troops into the chaos below, cracks and bloods riddle the walls of the old palace, cracks appear in the surrounding ground, symbolizing the crumbling foundation of the city's unity and tradition. Shadows of rebel forces emerge from the forested areas, strategically advancing under the cover of haze and destruction.

The palace's ancient architecture is both a relic of Tokyo's history and a symbol of its fragility, standing as a silent witness to the destruction wrought by the relentless conflict. The royal family remains trapped inside the palace, with nowhere to evacuate, they can only arm themselves and prepare for the worst.

The land of the rising sun faces its darkest hour, as the sun begins to set on the nation, Japan screamed into the night calling for a savior, and the light danced in her eyes far from reach….

1.5 Techniques Used

Techniques	Methods Used	Image
0.Original Image		
1. Color grading and tinting 2. Light and Shadow Manipulation	• Dimmed Image: The image brightness is reduced to 30% to create a darker, more unsettling mood. • Vignette Effect: -The corners are darkened, focusing attention toward the center and muting peripheral colors. Enhances the feeling of focus and isolation by darkening the image's edges..	
3. Overlaying Texture 4. Adding objects and elements	• Blood Stains: Random positions for blood stains are created and added to evoke violence that is happening in the diet. • Chalk Texture: The chalk texture of a human body is resized and overlaid on the image at a fixed position. It adds a physical, gritty feel to	

	the scene, showing the aftermath of a murder scene .	
5. Adding atmospheric effect 6. Creating composite images	•Alpha Compositing: Blood, chalk, and fog textures are layered over the base image using transparency, blending them seamlessly. • Fog and Image Combination: Fog is merged with the dimmed background, adding depth and obscured visibility for an eerie effect.	
7.Storytelling elements	•Blood Stains: Represents the aftermath of violence, enhancing the narrative. •Chalk Texture: Implies the scene of a recent murder, adding context to the violence. •Flash Effect with Alarm Sound: Flashing lights and an alarm sound suggest urgency or panic, implying an ongoing crisis. •Fog and Atmosphere: Adds mystery, making the scene feel unexpected and ominous.	

Technique	Methods Used	Image

0	Original Image	
1. Image Preprocessing (Darkening and Tinting) 2.Smoke Effect Circle Generation 3.Image Blurring	• Darkening and tinting the base image for mood setting • Circle Generation: Randomly places circles on a black canvas to simulate smoke. • Layer Blending: Multiple smoke layers are blended using cv2.addWeighted() to create a more intense, overlapping smoke effect. • Gaussian Blur (cv2.GaussianBlur()): Applied to the smoke, emergency lights, and final result to create a softer, more atmospheric look, simulating fog or haze and reducing harsh edges.	

4. Image Cropping, Resizing and Adjusting. 5.Storytelling Elements	• Rescaling with Aspect Ratio Preservation:Resizes and adjusts images to maintain the correct aspect ratio, ensuring natural scaling • Rescaling with Aspect Ratio Preservation: Resizes the riot crowd to maintain proportionality and creates a sense of tension by making the crowd appear larger and more dominant. • Alpha Channel Masking:Uses transparency to blend the crowd seamlessly with the background for immersion. • Colour Emphasis: Primarily uses red tones to heighten the atmosphere of violence and chaos.	

Technique	Methods Used	Image
0	Original Image	
1. Color Grading and Tinting 2.Light and Shadow Manipulation	• Desaturation (30% saturation) for a faded, bleak look. • Slight contrast boost for depth.	

	• Red tint for a dystopian feel.	
3.Adding Atmospheric Effect	-Fog texture added for a misty, ominous vibe.	
4.Overlaying Texture	• Cracks texture applied at fixed positions for damage. • Blood stains added to show violence and decay.	
5.Adding Objects and Elements 6.Creating Composite Images 7.Simple storytelling element	• Helicopter and rebel forces images added to show ongoing conflict at the Imperial Palace. • Combined fog, cracks, blood, and objects using alpha compositing to create a unified scene of violence and chaos. • helicopter descent with saved as a GIF to show movement.	

1.6 Screenshots of Solution

Overlaying the rioters and Scene Building at the National Diet

```
# Basic dark and red tint
darkened = cv2.convertScaleAbs(original_diet, alpha=0.6, beta=-40)
red_tint = darkened.copy()
red_tint[:,:,2] = cv2.add(red_tint[:,:,2], 40)
```

Figure 4: Method to add Red Tint and dystopian Filter

The purpose of the block of code above is to create a red, dystopian atmosphere in the original image of the National Diet, the cv2.convertScaleAbs() function darkens the original image by using a scaling factor of (alpha = 0.6) and adjusting the brightness (beta = -40), the red tint is further enhanced through the increase of the red_tint channel.

Figure 5: Original image with newly added Red Tint and Filter

The image here now has a red tint, which helps in the beginning of the creation of a dystopian and chaotic scene.

Figure 6: The image of the crowd to be overlayed

The background of the crowd is removed for better overlaying into the original image

```
# Create a canvas for the crowd that matches the main image width
crowd_canvas = np.zeros((crowd_height, width, 3), dtype=np.uint8)
crowd_mask_canvas = np.zeros((crowd_height, width), dtype=np.uint8)

# Place the crowd in the center of the canvas
crowd_canvas[:, x_offset:x_offset + crowd_width] = riots_red
crowd_mask_canvas[:, x_offset:x_offset + crowd_width] = crowd_mask

# Blend crowd using mask with increased opacity
crowd_region = result[y_offset:min(y_offset + crowd_height, height), :]
for i in range(min(crowd_height, height - y_offset)):
    alpha = crowd_mask_canvas[i] / 255.0 * 2
    alpha = np.clip(alpha, 0, 1)
    alpha = np.stack([alpha] * 3, axis=-1)
    crowd_region[i] = (1 - alpha) * crowd_region[i] + alpha * crowd_canvas[i]

result[y_offset:y_offset + crowd_region.shape[0], :] = crowd_region
```

Figure 7: Code to overlay the riot crowd onto the National Diet Building Image

The Crowd is temporarily placed into a canvas to hold the image file, after that alpha blending is used to apply varying levels of transparency to better blend the crowd into the background of the National Diet, the transparency of the crowd is then adjusted based on the mask of the crowd to create a more realistic and gradual overlay.

```python
def generate_smoke_layer():
    smoke = np.zeros((height, width, 3), dtype=np.uint8)
    num_smoke_points = 100
    for _ in range(num_smoke_points):
        x = random.randint(0, width-1)
        y = random.randint(0, height//3)
        radius = random.randint(20, 100)
        color = random.randint(150, 200)
        cv2.circle(smoke, (x,y), radius, (color,color,color), -1)
    return cv2.GaussianBlur(smoke, (99, 99), 30)

# Generate smoke effects
smoke_final = generate_smoke_layer()
for _ in range(3):
    smoke_final = cv2.addWeighted(smoke_final, 0.7, generate_smoke_layer(), 0.3, 0)
```

Figure 8: Code for Blending smoke layers into the final scene

Smoke layers are generated and combined with the addweighted() method to blend the original smoke with newly generated smoke values, the final smoke layer is added with blending to create the effect of continuous fog.

Figure 9: Image after blending smoke layers

As seen in the image above, the original image with the red tint is now enhanced with additional smoke layers and effects to better enhance the storytelling elements of the image, the smoke layers create a dark fog from the bottom to the middle of the image. Creating an even more unsettling scene of chaos and destruction about to come.

```
crowd_height = int(height * 0.4)
aspect_ratio = riots.shape[1] / riots.shape[0]
crowd_width = int(crowd_height * aspect_ratio)

if crowd_width < width:
    crowd_width = width
    crowd_height = int(crowd_width / aspect_ratio)
    if crowd_height < int(height * 0.4):  # Ensure minimum height
        crowd_height = int(height * 0.4)
        crowd_width = int(crowd_height * aspect_ratio)

riots_resized = cv2.resize(riots, (crowd_width, crowd_height))
x_offset = (width - crowd_width) // 2
```

Figure 9: Code for adjusting crowd size and centres them in the image

The size of the crowd is calculate through maintaining the original aspect ration, this is so that the crowd can be dynamically fit into the scene, cv2.resize() is used to adjust the crowd size with x_offset to centre the crowd horizontally.

Figure 10: Image after blending smoke layers

This is the final product of all the techniques used for this scene. A large crowd of rioters is overlaid onto the National Diet scene with red tint and smoke. Emergency lights and flashing red lights enhance the chaotic atmosphere.

Combining elements, randomized elements, Sound Effect and Flashing Lights to form a chaotic Image inside the National Diet

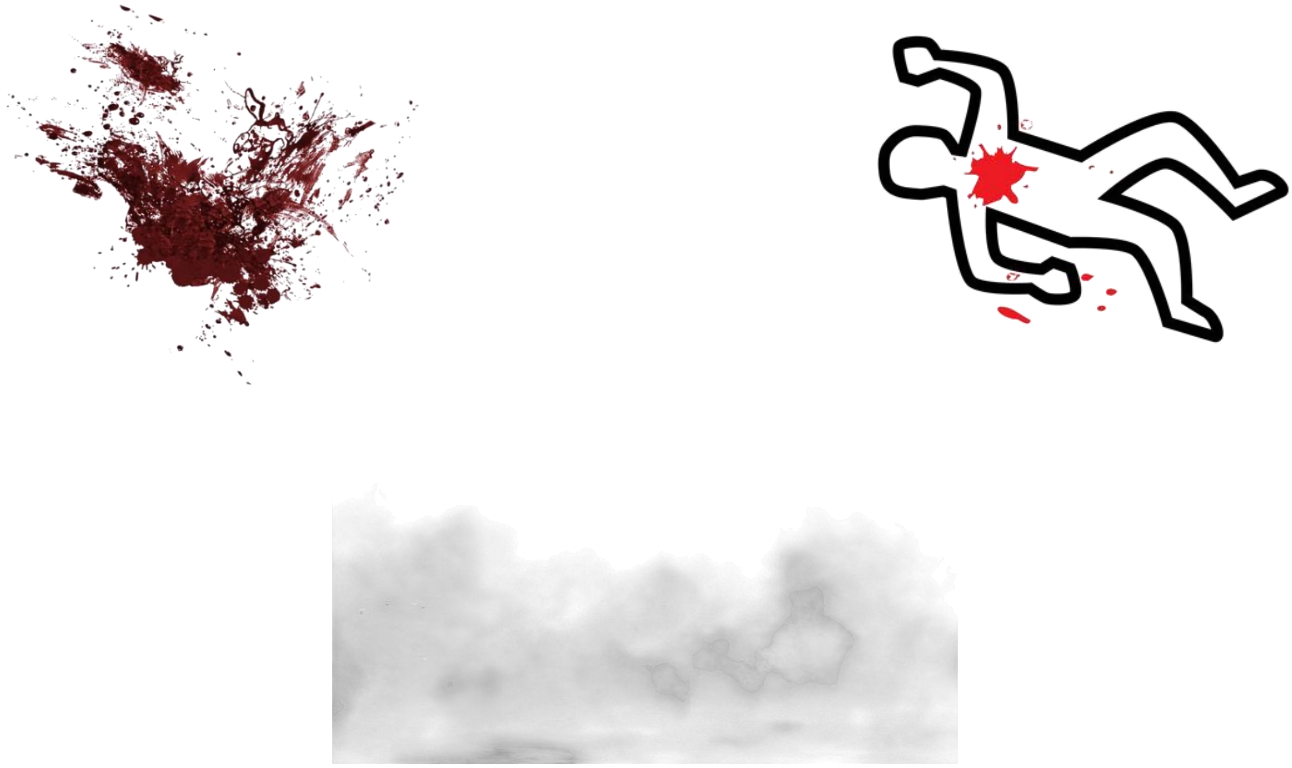


Figure 11: Elements used for the scene inside the National Diet

The elements here are used to illustrate the scene of chaos in the National Diet immediately after the assassination of the Prime Minister, the fog is used to set the eerie and tense scene in the diet, the chalk outline of the dead body will be used to show the Prime Minister lying on the ground dead, and the blood spatter will be spread around the vicinity of the body build the scene.

```python
def generate_blood_positions(center_x, center_y, num_stains=5, spread_radius=100):
    positions = []
    for _ in range(num_stains):
        offset_x = random.randint(-spread_radius, spread_radius)
        offset_y = random.randint(-spread_radius, spread_radius)
        new_x = max(0, min(center_x + offset_x, dimmed_image.width - 100))
        new_y = max(0, min(center_y + offset_y, dimmed_image.height - 100))
        positions.append((new_x, new_y))
    return positions
```

Figure 12: Randomised function for generating blood splatter

The function generates a set of random positions around a central point (center_x, center_y). It ensures the positions of the bloodstains stay within the bounds of the image by clamping values to the image dimensions.

```python
chalk_resized = chalk_image.resize((100, 100))
chalk_position = (621, 590)
background.paste(chalk_resized, chalk_position, chalk_resized.convert('RGBA').split()[3])

fog_texture_with_reduced_opacity = reduce_fog_opacity(fog_texture, opacity=0.5)
fog_texture_resized = fog_texture_with_reduced_opacity.resize(background.size)
```

Figure 13: function for adding chalk outline and fog overlay

The chalk body outline is imported and then positioned in a specific coordinate in the image, the fog overlay is added previously and has its opacity being reduced to improve visibility in the image.

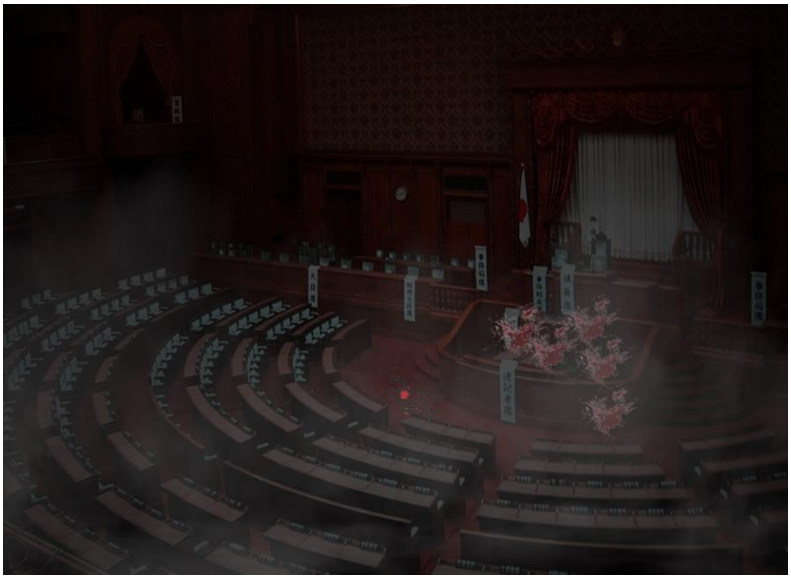


Figure 14: Composite image combining all previous elements

The image shown here is the composite image after adding all the elements mentioned, the fog can be seen covering the entirety of the diet hall, the body outline of the prime minister lying in the centre of the hall along with bloodstains spread around the stage of the diet.

```
def apply_flash_effect(image, flash_color=(255, 0, 0), flash_duration=0.5, num_flashes=10):
    image = image.convert("RGBA")
    flashes = []

    for _ in range(num_flashes):
        overlay = Image.new('RGBA', image.size, flash_color + (0,))
        flashed_image = Image.blend(image, overlay, alpha=0.4)
        flashes.append(flashed_image)
        flashes.append(image)
        time.sleep(flash_duration)

    return flashes

def play_alarm_sound():
    alarm_sound = pygame.mixer.Sound("alarm.wav")
    alarm_sound.set_volume(0.2)
    alarm_sound.play(loops=-1, maxtime=5000)
    return alarm_sound
```

Figure 15: Adding Flashing lights and alarm sound into the final image

The code creates multiple "flash" frames where the original image is blended with a red-colored overlay (flash_color), creating the flash effect simulating the alarm in the Diet Hall going off. The flashes are then alternated with the original image.The time.sleep(flash_duration) ensures that the

flashes occur at timed intervals, the flash is also enhanced with sound with the use of an audio file to enhance immersion

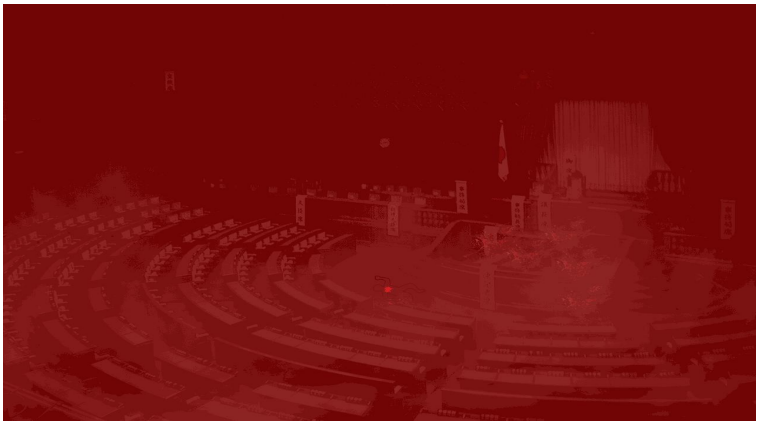


Figure 16: Final image with Flashing lights and alarm sound and composite images

Shown here is the final product of the enhancements mentioned previously.

Adding hovering helicopter animation carrying Corrupt government forces moving towards the Imperial Palace

Figure 17: Helicopter image used to create the animation

The background of the helicopter is first removed to ensure better overlay in the image

```python
def add_helicopter_to_frame(base_img, heli_img, position):
    # Ensure base_img is a PIL Image (RGBA format)
    if isinstance(base_img, np.ndarray):
        base_img_pil = Image.fromarray(base_img).convert('RGBA')
    else:
        base_img_pil = base_img.convert('RGBA')

    # Paste the helicopter image at the specified position with transparency handling
    base_img_pil.paste(heli_img, position, heli_img)  # Use heli_img as the mask for transparency
    return base_img_pil

# Function to create an animated GIF with the helicopter lowering down
def create_animated_helicopter_gif(base_img, heli_img_path, start_pos, end_pos, gif_path, swat_img, swat_position
    # Load the helicopter image
    heli_img = Image.open(heli_img_path).convert('RGBA')

    # Resize the helicopter to be 50% smaller
    heli_size = tuple([int(dim * 0.5) for dim in heli_img.size])  # Make it 50% smaller
    heli_img = heli_img.resize(heli_size)

    # Create two frames: one with the helicopter at start_pos, and one at end_pos
    frames = []

    # First frame (helicopter at start position, add SWAT)
    frame1 = add_helicopter_to_frame(base_img, heli_img, start_pos)
    frame1_with_swat = add_helicopter_to_frame(frame1, swat_img, swat_position)  # Add SWAT at the same time
    frames.append(frame1_with_swat)

    # Second frame (helicopter at end position, add SWAT)
    frame2 = add_helicopter_to_frame(base_img, heli_img, end_pos)
    frame2_with_swat = add_helicopter_to_frame(frame2, swat_img, swat_position)  # Add SWAT at the same time
    frames.append(frame2_with_swat)

    # Save the frames as an animated GIF
    frames[0].save(gif_path, save_all=True, append_images=frames[1:], optimize=False, duration=1000, loop=0)

# Define the helicopter start and end positions
start_position = (290, 100)  # Initial position of the helicopter
end_position = (330, 150)    # Lowered position of the helicopter
```

Figure 18: Code for loading and animation helicopter image

The code defines two functions, one for creating the GIF of the helicopter hovering and lowering down, the add_helicopter_to_frame function overlays the helicopter onto the original image in the specified coordinates, the base image at a specified position, handling transparency. The create_animated_helicopter_gif function generates the animation placing the helicopter at two positions (start and end), creates two frames with the helicopter at different positions, and combines them into an animated GIF.

Figure 19: Helicopter hovering overlay in the final enhanced image

The addition of the helicopter GIF animation completes the entire scene of chaos occurring at the Imperial Palace.

Scene 3: City Nuked by Enemies

1.7 Storyline and Images

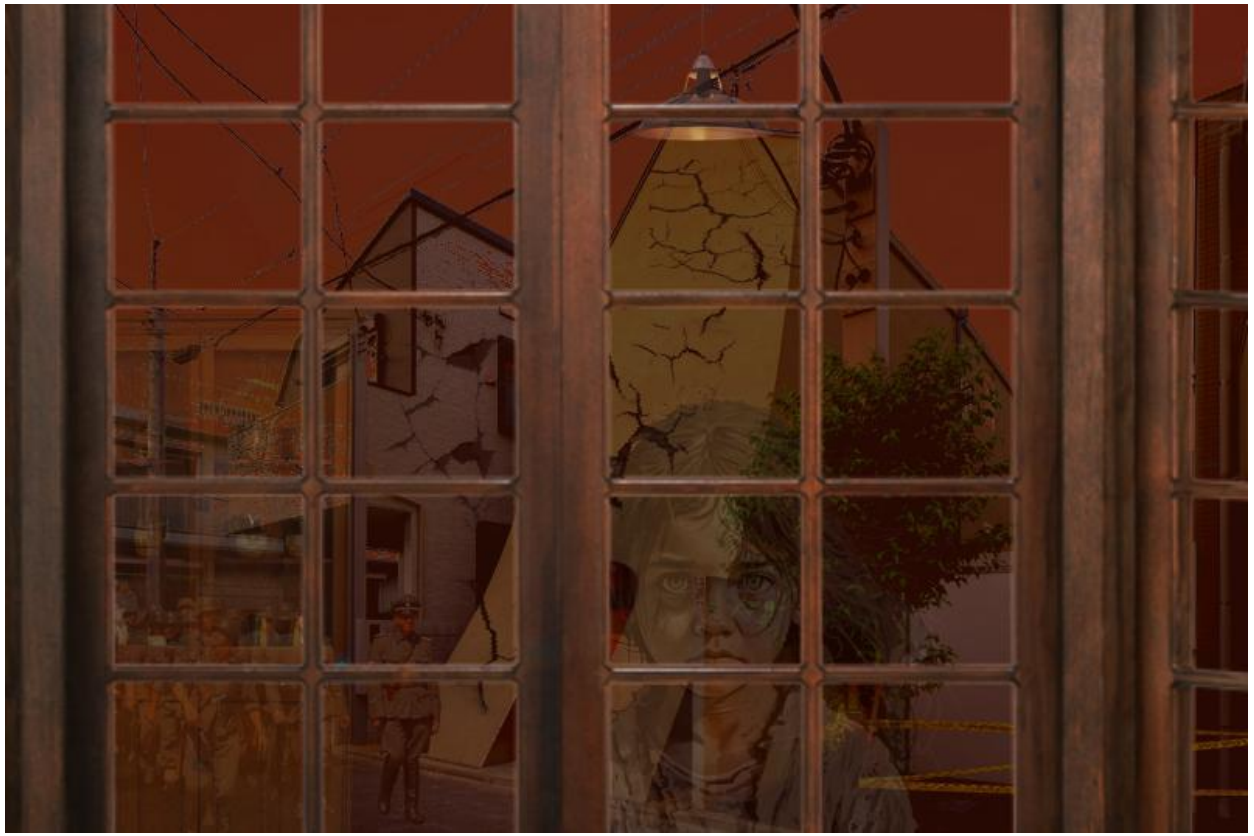


Figure 1 Scared girl staring out from the window at the streets occupied by opposing armies.

Due to the corruption, city defences and city infrastructure weakened quickly over time. This formed an opening for the long-awaiting opposing countries. The armies of the opposing countries invaded the city and took control of the streets of Tokyo. Citizens of the city were forced to stay locked in their homes, afraid to come out. Children were unable to attend school as usual, and only dared to stare out of their windows looking at the unwilling scenery of the degrading streets and houses, afraid that going out will threaten their lives.

Figure 2 The city getting bombed, turning into living hell.

Suddenly on a bright sunny day, some fighter aircrafts appeared in the skies of Tokyo. The aircrafts dropped bombs and fled off. Multiple locations of the city of Tokyo were bombed, engulfing the city in flames. The dirty streets suddenly seemed so peaceful, only to be took over by flames and screams in a split second. It became a living-hell for the civilians of Tokyo.

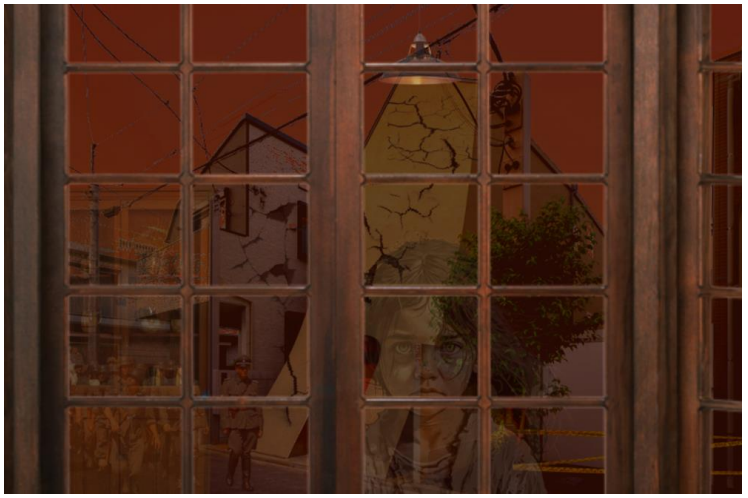
Figure 3 Neighbouring streets of bomb sites was quickly filled with smoke and lands cracked

The city of Tokyo was quickly filled with toxic fumes, and civilians fled for their lives. Streets near to the location of bombings, had their lands cracked due to the immense vibrations form the explosions. Other infrastructure such as the digital screens also gave in to the disturbance and broke down, with the streets looking gloomy, and vison was impaired due to the thick smoke from the burns from neighbouring bombsites.

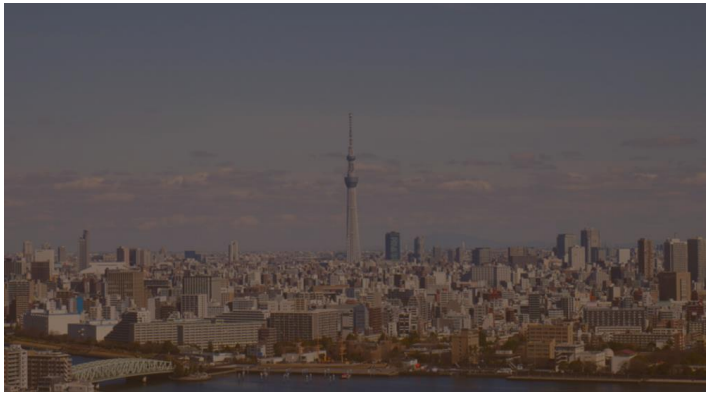
1.8 Techniques Used

Technique	Methods used	Images
0	Original image	
1. Adding objects and elements	• Creating the objects (patrolling troops, wall crack 1,2 &3, police line) • Decreasing brightness of objects to match the scene • Resizing objects • Rotating objects • Pasting onto base image	

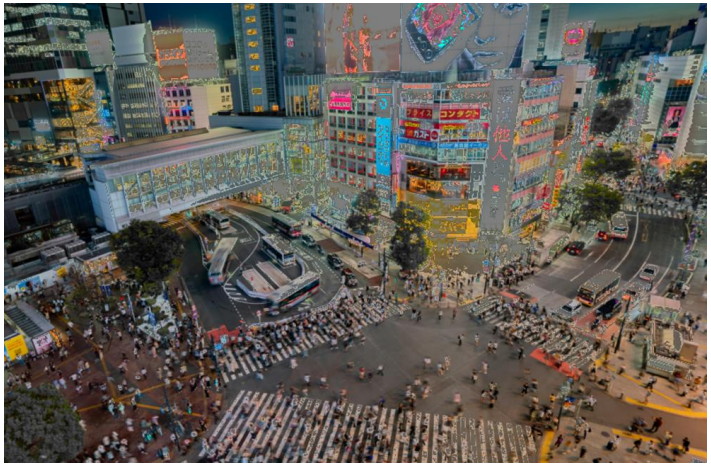
2. Light and Shadow Manipulation	• Masking the sky using blue color ranges • Creating burning red filter • Blending the red filter into the sky using the sky mask created	
3. Colour Grading and Tinting	• Enhance contrast of image • Enhance saturation • Creating a gray-scale version of original image, then converting the black and whites to create a sepia filter • Blending the sepia filter into the image • Enhancing brightness	

4. Creating storytelling effect	• Create a transparent background • Adding the objects onto the transparent background (girl, ceiling light, and cupboard) with reduced transparency and brightness to match final scene image • Pasting the window frame on top to simulate the reflection of the girl from the inside	
5. Creating Composite Images	• Pasting the created window reflection illusion onto the outside scenery • Adjust transparency to improve realistic reflection	

Technique	Methods used	Images
0	Original Image	
1. Adding objects and elements	• Adding objects (jet fighter and bomber plane) • Resizing the objects • Rotating the objects • Pasting the objects onto the image	

2. Colour Grading and Tinting	• Decreasing the brightness • Created reddish-orange filter • Blended into base image with adjusted transparency	
3. Overlaying Textures & Creating composite image	• Creating mask for the buildings leaving the facade of the buildings • Creating the hell-burn background • Reducing brightness of the background • Applying Gaussian Filter to blur the background • Pasting the background to the images excluding the buildings mask	 

4. Adding Objects and Elements	• Adding the explosion ‘mushroom’ • Resized the object • Reduced brightness of explosion to blend into the image	
5. Adding Atmospheric Effects	• Adding thick smoke	
6. Storytelling elements	• Adding animation for the bomb getting dropped onto the city • Adding animation for the bomb explosion • Adding animation to show fire sparks rising on the hellish-background image of the city after getting bombed • Flashing between the peaceful original image and disaster bombed city to create tense and stressful feelings. Also creates emphasize the comparison between peaceful and disaster	

Technique	Methods used	Images
0	Original Image	
1. HDR Effect	• Masking the brightly lit digital screens • Reducing the brightness of the screens to make it look like they are broken	

2. Adding objects and elements	• Adding a land crack • Flipping the land crack • Resizing the land crack to become bigger • Rotating the land crack • Adjust brightness and reduced transparency to fit the scene	
3. Adding texture	• Added noise into the picture to simulate dust	
4. Adding Atmospheric Effects	• Added thick smoke overlay • Adjusting brightness and saturation to allow it to blend smoother into the scene	

5. Colour Grading and Tinting	• Decrease overall brightness • Adding orange filter to add apocalyptic atmosphere	

1.9 Screenshots of Solutions

Creating the reflection of the girl looking outside from her house in the window

```
# Create the illusion of a girl looking outside from her house by putting a  girl's reflection in the window
window = Image.open("photos/window.png").convert("RGBA")
window = window.resize(street.size)

window = window.crop((200,150,window.width-150,window.height-70))
window = window.resize(street.size)

girl = Image.open("photos/girl.png").convert("RGBA")
girl.thumbnail((300,300))
enhancer = ImageEnhance.Brightness(girl)
girl = enhancer.enhance(0.4)
girl_silhoutte = girl

reflection_alpha = girl.split()[3]  # Get the alpha channel (transparency)
reflection_alpha = reflection_alpha.point(lambda p: p * 0.7)  # Decrease the opacity
girl.putalpha(reflection_alpha)

hangingLight = Image.open("photos/hanging light.png").convert("RGBA")
hangingLight.thumbnail((100,100))
hangingLight = hangingLight

reflection_alpha = hangingLight.split()[3]  # Get the alpha channel (transparency)
reflection_alpha = reflection_alpha.point(lambda p: p * 0.8)  # Decrease the opacity
hangingLight.putalpha(reflection_alpha)

cupboard = Image.open("photos/cupboard.png").convert("RGBA")
cupboard.thumbnail((400,400))
enhancer = ImageEnhance.Brightness(cupboard)
cupboard = enhancer.enhance(0.6)

reflection_alpha = cupboard.split()[3]  # Get the alpha channel (transparency)
reflection_alpha = reflection_alpha.point(lambda p: p * 0.7)  # Decrease the opacity
cupboard.putalpha(reflection_alpha)

reflection_in_window = Image.new("RGBA", window.size, (0, 0, 0, 100))
reflection_in_window.paste(girl, (320, (reflection_in_window.height-girl.height)), girl.split()[3])
reflection_in_window.paste(hangingLight, (400, 0), hangingLight.split()[3])
reflection_in_window.paste(cupboard, (0, (reflection_in_window.height-cupboard.height+50)), cupboard.split()[3])
reflection_in_window.paste(window, (0, 0), window.split()[3])
display(reflection_in_window)
street.paste(reflection_in_window, (0, 0), reflection_in_window)
```

Figure 4 Code used to create reflection in window.

Above is the solution used to adding a scared girl's reflection on a window, creating the illusion that the girl is looking at the outside from within her house.

Figure 5 Indoor elements used to create the indoors reflection on the window

First, images of indoor elements are imported. Above figure shows the elements used: a cupboard, and indoor ceiling hanging light, and the main subject- the scared girl. These objects have their brightness and transparency adjusted in order to make them look like reflections on the window's glass panels. The hanging light and cupboard images are used to hint that the girl is in her home, looking at the outside.

Figure 6 The original image of the window frame used.

After the objects are ready and adjusted to fit the scene. A semi-transparent black canvas is used, and the three elements are pasted at specific locations. The canvas used is black in order to simulate the presence of glass panels of the window. Then, the image of a window frame as shown in the figure above is imported, cropped, and resized. The window is then pasted over the three elements, so that the reflections are only shown on the transparent glass panels, and not on the wooden window frames. This creates a more realistic illusion of reflections on the window glass panels.

Figure 7 Output of the window with reflections.

Above is the final 'window with reflections' image produced. This image is then pasted over the streets image to complete the composite scene.

Replacing image background with a hell-looking image

```
# GET BUILDINGS MASK OF THE TOKYO IMAGE
tokyo = cv2.imread('photos/cityscape of tokyo.jpg')
gray = cv2.cvtColor(tokyo, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, threshold1=50, threshold2=150) # Apply edge detection (e.g., Canny edge detection)
kernel = np.ones((5, 5), np.uint8) # Use dilation to make the mask solid (fills gaps)
mask = cv2.dilate(edges, kernel, iterations=1)
mask = cv2.bitwise_not(mask) # Invert the mask so that buildings are white and sky is black
mask_pil = Image.fromarray(mask) # Convert the mask to a format usable by PIL
mask_pil.save('output/buildings_mask.png') # Save the mask if needed

# Create the apocalyptic background
hell_background = Image.open('photos/hell_bg2.webp').convert("RGBA")
hell_background = hell_background.resize(hell_tokyo.size)
  # blur the background
hell_background = hell_background.filter(ImageFilter.GaussianBlur(radius=2))
  # Reduce brightness of bg
enhancer = ImageEnhance.Brightness(hell_background)
hell_background = enhancer.enhance(0.5)
  # Add the mushroom to background
mushroom = Image.open('photos/bomb (3).png').convert("RGBA")
mushroom = mushroom.resize((300, 300))
  # Reduce brightness
enhancer = ImageEnhance.Brightness(mushroom)
mushroom = enhancer.enhance(0.5)
mushroom = mushroom.filter(ImageFilter.GaussianBlur(radius=1))
hell_background.paste(mushroom, (100, 15), mushroom)
  # Apply apocalyptic backgroun without masked buildings
buildings_mask = Image.open('output/buildings_mask.png').convert("L")
buildings_mask = buildings_mask.resize(hell_tokyo.size)
hell_tokyo = Image.composite(hell_background, hell_tokyo, buildings_mask)
```

Figure 8 Solution used to achieve replacement of background using molten hell-looking image.

The above shows the code used to create a mask to separate the buildings from the background. The mask is then used to replace the original image's background.

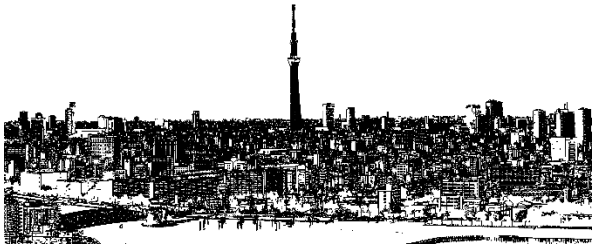


Figure 9 Resulting mask of buildings and background created.

First, the original image is imported, then a mask of the buildings and background are created using OpenCV's dilate function. The above figure shows the resulting mask, separating the building from the background, and removing some walls of the buildings to create the illusion of broken/burned down structures.

Figure 10 replacing the background with molten lava iamge.

Then, the hell molten lava background is imported. Adjustments is then made to the background by reducing brightness and blurring the image using a Gaussian Blur filter from the Pillow library. An explosion mushroom element is also added according to the bombing location depicted in the previous image. Finally, the background is pasted onto the base image by using the mask created previously to avoid the buildings.

Creating animations of the rising flame sparks and flashing-change of scenes

```
# ADD THE SPARKS ANIMATION AND FRAME CHANGING ANIMATION
ori_tokyo = Image.open('photos/cityscape of tokyo.jpg').convert("RGBA")
ori_tokyo.thumbnail((800,800))

num_frames = 60  # Number of frames in the animation
vertical_shift_per_frame = -5  # Pixels to move up each frame
amplitude = 10  # Amplitude for horizontal wiggle
frequency = 0.2  # Frequency for horizontal wiggle

i = 0
while i < num_frames:
    frame = hell_tokyo.copy()
    disaster_tokyo_interval = random.randint(8, 13)

    # Check if it's time to show the special image
    if i % disaster_tokyo_interval == 0:
        # Generate a random number of frames (between 1 and 2)
        ori_tokyo_duration = random.randint(1, 2)
        for _ in range(ori_tokyo_duration):
            frame.paste(ori_tokyo, (0, 0), ori_tokyo)
            frames.append(frame)

        # Increment `i` by the random duration to skip to the next segment
        i += ori_tokyo_duration
    else:
        # Regular animation: Calculate the vertical and horizontal shift
        vertical_shift = i * vertical_shift_per_frame
        if vertical_shift >= fireSpark.height - hell_tokyo.height:
            vertical_shift = vertical_shift % (fireSpark.height - hell_tokyo.height)

        horizontal_shift = int(amplitude * math.sin(frequency * i))

        # Apply transformation to create the offset effect
        transformed_sparks = fireSpark.transform(
            fireSpark.size,
            Image.AFFINE,
            (1, 0, horizontal_shift, 0, 1, -vertical_shift),
            resample=Image.NEAREST
        )

        # Crop the transformed sparks to the base image size
        cropped_sparks = transformed_sparks.crop((0, 0, hell_tokyo.width, hell_tokyo.height))

        # Paste the cropped, shifted sparks onto the frame
        frame.paste(cropped_sparks, (0, 0), cropped_sparks)

        # Append the regular animation frame to the list
        frames.append(frame)

        # Increment frame counter
        i += 1
```

Figure 11 Code used to creating the flame spark and flashing scene animation.

Above is the code used to create a storytelling effect, showing flame sparks engulfing the city and rising upwards. The animation also includes the flashing changes between the peaceful city and the hellish city, to create a stressful comparison between the before and after scenarios of the city getting bombed.

Whereas the flash-changing between the peaceful and disastrous scenes of the city is done by using different intervals for the peaceful scene, and the disastrous scene. The duration of showing each scene is controlled by using random intervals, with the disastrous scene having a longer interval compared to the peaceful scene. Then, the scenes will be displayed one after another, until each frame counter has reached the intended duration. The intervals between each change of scene are randomized, to simulating a stressful storytelling feeling for viewers. This also creates an emphasis of the difference of the city before and after the bombing effect.

When the disastrous scene of the city is displayed, the illusion of flame sparks rising is done by using a vertical and horizontal shift. The vertical shift is uniform for each frame, whereas the horizontal shift uses a random value to replicate the random swaying of flame sparks from left to right in the real world. Transformation is then applied to the sparks according to the vertical and horizontal shifts to create the animation.

Scene 4: Post Apocalypse

1.10 Storyline and Images

Tokyo is oddly quiet, unlike the bustling capital city that it was. The vending machine over there – used to be filled with drinks and snacks for walkers rushing by – are shrivelled now, colours faded with dirt and grime. Without power, they are nothing but empty shells, the husks of a city that was once moving and breathing. Concrete is cracked with weeds and other plants taking root squatting between the gaps of what was concrete where people once walked and gathered. It's as though nature is slowly reclaiming that which once was paved over with concrete and steel

The ridge of the array visible from the hilltop was lined with abandoned poorly maintained buildings. Windows are broken or absent, and even vines creep down onto roofs from fences, backdrops. Neglect has allowed green moss and grass slowly reclaim these surfaces, rounding the sharp edges of the city. A stop signs rusts nearby, although who it is meant for is a matter of some debate. No vehicles in the road, no walkers, just stillness. They paint an image of desolation where time seems to have frozen. Over the years nature has crept in, extended its roots and branches around the areas people once occupied.

A decrepit old box of shopfront around the corner, where colourful signs proclaimed vibrant business now long gone. Metal shutters are dropped, rusted and caked in decades of grimes, with no indication of anyone trying to swing them open. Faded, its sign now dangling drunkenly from above, letter bold but flaked throughout so that the words are hard to read. While this was probably a local shop or market, it left the public eye long ago. The streets are strewn with a massive pile of rubble, the debris from at least one nearby collapse, proof of how forgotten the city has truly become.

Trees and plants have started to grow unchecked, their roots pushing through the cracks in the road and reaching out onto the sidewalks. A deep hole in the ground disrupts the road entirely, possibly the result of an explosion or simply decay over time. The pavement is cracked and uneven, littered with debris from fallen buildings and overgrown plants. The entire area looks frozen in a state of abandonment, left to deteriorate with no one around to care for it. Nature is slowly overtaking what was once a populated neighbourhood, filling the empty spaces and reclaiming the land that had been transformed into a city.

1.11 Techniques Used

Technique	Methods used	Sample Images
	• Original Picture	
Color Grading Tinting	The top layer of the picture is subjected to colour grading, which results in an overall dull and dark tone often associated with deserted places. By decreasing the colour saturation, the lively colours are lost, resulting in an old and flat looking scene. In addition, more detailed changes to the brightness and contrast serve to highlight shadows and gives depth to the image, as well as the feeling of neglect. The image is softened by a 2 pix Gaussian blur, providing a slight fog that adds to the post-apocalyptic effect, and indicates dirt and dust accumulation over time.	

Overlaying Texture	In order to represent the process of nature taking over the city, several textural aspects such as moss, dust, and cracks have been placed on top of the primary image. The code resizes, blurs, and modifies brightness, contrast, and colour for each overlay, so that it is not out of place with the surrounding environment. Such attention to detail on textures enables the moss to appear as though it is growing on the surfaces of the walls, rather than painted on. The cracks too appear not as designs, but as natural cracks brought upon by uniform decay. Every texture is placed carefully to enhance the areas that would naturally attract certain elements or damage thus making the image more believable.	
Adding Object and Elements	In this depiction, trees, shrubs, and other objects are also included in the urban build up depicting the encroachment of plants within cities. Every single component is controlled in its position by a set of coordinates (for example, position in the overlays specs) and adjusted to scale in the perspective of the image in the background. This accurate placement of elements assists in developing a storyline of great reclamation whereby the deathly city scape is slowly filled with greenery. Some elements are also rotated (using rotation parameters) allowing for some variety and giving a more realistic touch as if plants have grown on their own rather than being placed in specific uniform positions.	

Light and Shadow Manipulation	In order to ground all overlay components within and in the given environment, shadows are incorporated into every overlay component. The creation of shadows involves firstly converting each overlay into a black and white dull image and stylizing with colors black and grey in order to imitate shadows. By putting an opacity level to the shadow, the code creates a smooth and realistic looking shadow that slightly shifts each element. This position makes the overlay components appear like they are throwing shadows over the structure or the floor, which adds to the perspective and realism	
Creating Composite Image	Thereby painting the final image would requires layered rendering of different components until reaching the wish image base. Initially, shadows are imposed and then parts that should be overlaid so that the shadow effect fills in below the items it decorates. Alpha transparency is used so that every overlay fits into the base image seamlessly – all added textures and elements tend to act as naturally part of the image instead of distinct layers on surface. Such composite techniques enable several illustrations of moss, crack, and vegetation to co-exist and overlap produce a more realistic scene.	

Technique	Methods used	Sample Images
	• Original Picture	
Color Grading Tinting	Following the application of the HDR effect, color grading is employed in order to reduce saturation further and effectively create a duller and sober tone. The study brightens the level of contrast while reducing the brightness of the colors to assist in the enhancement of shadows as well as overall depth. This color grading makes the image seem like new and old, as if, over the course of time, several layers of dust and decay have been allowed to settle on the photograph. The gentle softening gives a damp, lonely atmosphere to the photo, emphasizing the sense of the place being lost, and forgotten.	

HDR Effect	To enhance the textures and details of the base image layer, and HDR effect is overlaid, making every element in the scene more dramatic and aged. This is done by applying Contrast Limited Adaptive Histogram Equalization (CLAHE) on L-channels in LAB colour space. LAB is useful in this scenario as it allows for the separation of luminance and color making it easy to enhance where needed. The use of this HDR effect moreover increases to emphasize the details in risky manner of enhancement where contrasting texts become easily visible as where such house moss features or surfaces are protruded and stand out cleanly – which is ideal for post-apocalyptic images.	
Overlaying Texture	Different textures such as peeling surface, moss, and dirt are used as overlays to depict the degradation of urban structural elements over time. Each overlay has been resized, slightly blurred, and adjusted with respect to brightness, contrast, and colour for effective integration in the scene. The code simulates a distant, overgrown scene filled with crack, moss, and dirt by layering intricate details. The overlays are arranged in the way that the mock-up depicts realistic decay patterns, such as that of the moss growing on the surfaces of the wall or cracks forming on the road.	

Adding Object and Elements	To emphasize the delaying reintroduction of nature in the city, other elements such as foliage and ground moss, are included in the design. The location and dimensions of each Overlay correspond to the viewpoint and scale of the background image. Certain overlays are also rotated for variation and realism. This technique enhances the scene with detail and adds more dimensionality as if nature is taking over the neglected areas of the city.	
Light and Shadow Manipulation	For every overlay added, shadow is made in order to provide a sense of depth and realism to the objects embedded in the scene. Specifically, in each overlay a shadow is created, and involves turning the overlay into black, and then moving it backwards slightly to create a shadow. These features reinforce the three-dimensional effect thus making the overlays convincingly part of the surrounding environment. This level of detail guarantees that every crack, moss, or bush looks like it belongs to the picture background.	

1.12 Screenshot of solution

HDR Effect

```python
def apply_hdr_effect(image):
    img_np = np.array(image.convert("RGB"))  # Convert to RGB for processing
    lab_img = cv2.cvtColor(img_np, cv2.COLOR_RGB2LAB)
    l_channel, a_channel, b_channel = cv2.split(lab_img)

    # Apply CLAHE to L channel
    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8, 8))
    l_channel = clahe.apply(l_channel)

    # Merge back to LAB and convert to RGB
    lab_img_clahe = cv2.merge((l_channel, a_channel, b_channel))
    hdr_img_np = cv2.cvtColor(lab_img_clahe, cv2.COLOR_LAB2RGB)
    return Image.fromarray(hdr_img_np)

base_image_hdr = apply_hdr_effect(base_image)
```

To enhance the textures and details of the base image layer, and HDR effect is overlaid, making every element in the scene more dramatic and aged. This is done by applying Contrast Limited Adaptive Histogram Equalization (CLAHE) on L-channels in LAB colour space. LAB is useful in this scenario as it allows for the separation of luminance and colour making it easy to enhance where needed.

Overlaying Textures and Adding Object Elements

```python
for spec in overlays_specs:
        overlay_image = spec['image']
        overlay_resized = overlay_image.resize((spec['width'], spec['height'])).filter(ImageFilter.GaussianBlur(1))

        # Apply enhancements
        overlay_resized = ImageEnhance.Brightness(overlay_resized).enhance(spec.get('brightness', 0.4))
        overlay_resized = ImageEnhance.Contrast(overlay_resized).enhance(spec.get('contrast', 0.7))
        overlay_resized = ImageEnhance.Color(overlay_resized).enhance(spec.get('color', 1.5))
```

```python
{
        'image': overlay_images[8], #moss8
        'position': (int(base_image.width * 0.60), int(base_image.height * 0.55)),
        'width': int(base_image.width * 0.15),
        'height': int(base_image.height * 0.2),
        'brightness': 0.7,
        'contrast': 1.5,
        'color': 0.2
    },
```

In order to represent the process of nature taking over the city, several textural aspects such as moss, dust, and cracks have been placed on top of the primary image. The code resizes, blurs, and modifies brightness, contrast, and colour for each overlay, so that it is not out of place with the surrounding environment. Every single component is controlled in its position by a set of coordinates (for example, position in the overlays_specs) and adjusted to scale in the perspective of the image in the background. This accurate placement of elements assists in developing a storyline of great reclamation whereby the deathly city scape is slowly filled with greenery.

Scene 5:

1.13 Storyline and Images

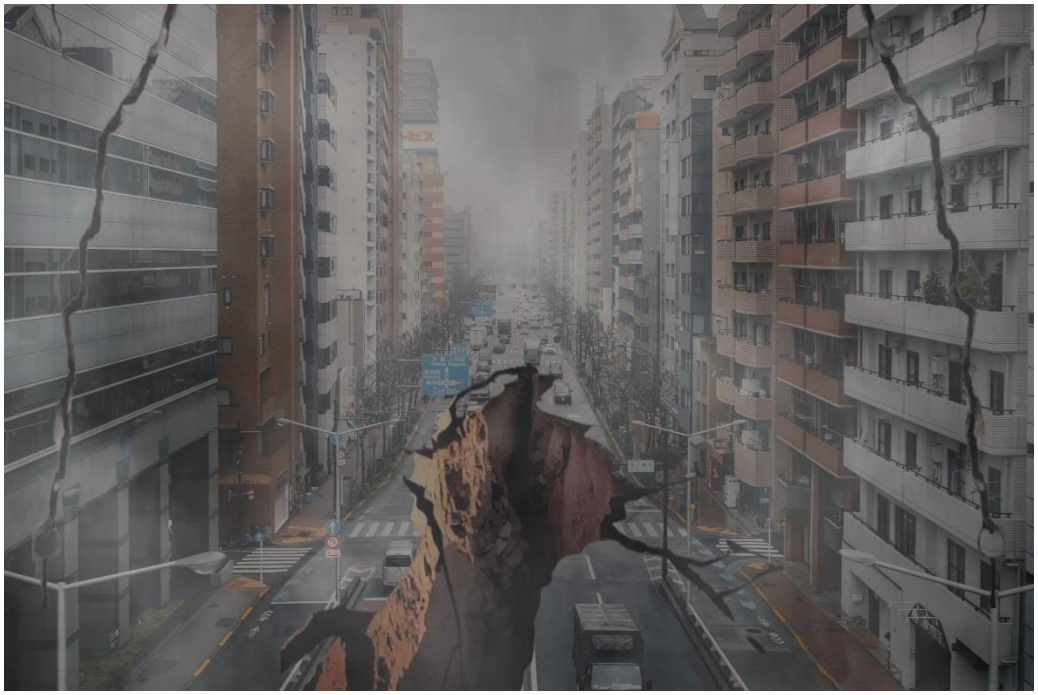


Figure 12: Poisonous Fumes Settle Down

Finally, it seems that the dense, oppressive cloud of toxic gases that previously covered Tokyo's streets is starting to clear. Even if it still has a slight bitter Odor, the air no longer chokes anything that dares to breathe. Roads are divided by sharp fissures that like veins of destruction, and buildings are pocked with cracks that run through façade. Automobiles left in the midst of mayhem are stuck in place, some buried under rubble and others sitting eerily unaltered. The once vibrant and busy city now reverberates with a disquieting quiet, interrupted only by the distant flutter of settling ash or the sporadic moaning of crumbling buildings. The first rays of sunlight struggle through the remaining haze as the skies gradually clear. Tokyo stands as a silent testament to survival—a wounded yet unbroken giant waiting for life to return.

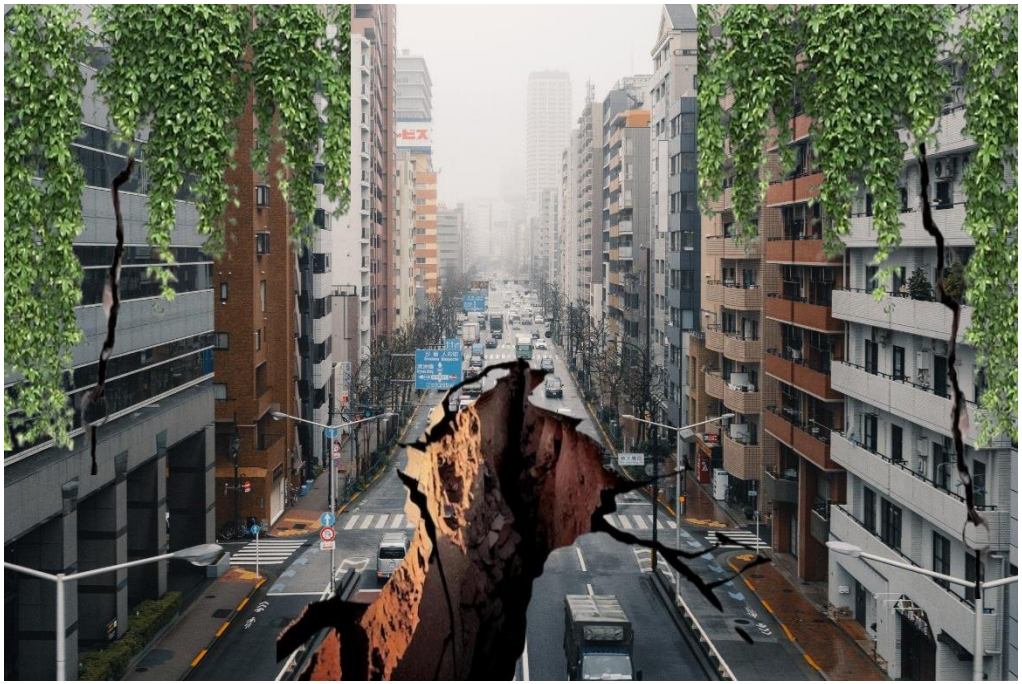


Figure 13: A Glimpse of Life

Although being broken and damaged, Tokyo's streets start to show hints of resiliency. Tiny vines droop from building crevices, their vivid green a sharp contrast to the bleak emptiness. Persistent and unyielding, nature claims what remains of the world created by humans. Birds start to return in the midst of the quiet, their chirping resonating through the deserted city halls. The air has a faint but unmistakable vibrancy, as though Tokyo is reviving. Patches of warmth appear on the eroded roads as the sunlight increasingly penetrates the remaining haze. The few who dared to return venture cautiously into the streets, their footsteps tentative yet hopeful. Some carry tools, while others gather whatever is salvageable from the debris. Their faces, though weary, reflect a quiet determination. Life, fragile but unyielding, has begun to reemerge from the ashes.

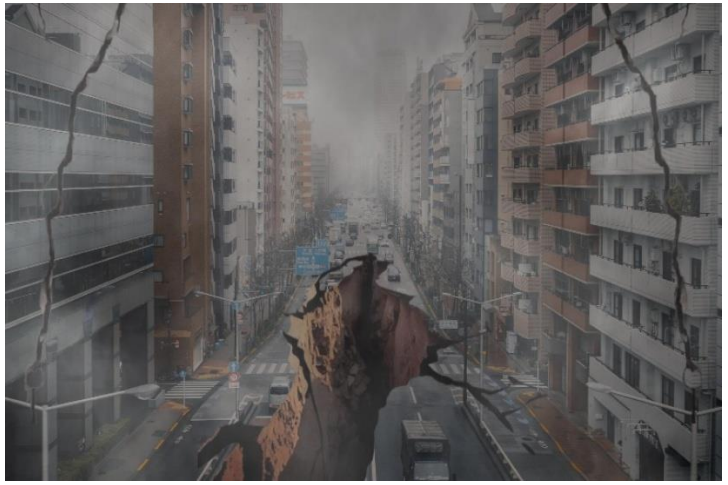
Figure 14: Rebuilding Their Home

The sound of progress fills the city. Employees wearing helmets and bright vests line the streets, their presence a testament to tenacity and hope. Trucks remove rubbish to make room for new foundations, while heavy machinery roars and hums, moving dirt and rubble. Road and building flaws are now viewed as challenges to be overcome rather than scars. Volunteers and families get together to sort through the remnants of their homes. Voices fill the room, some instructing, some laughing, others offering consolation. Little green areas that had begun to reclaim the city earlier are now being meticulously tended to and are becoming a feature of the new Tokyo. Every act of rebuilding, no matter how small, speaks of resilience and unity. The city, once brought to its knees, begins to stand tall again, brick by brick, hope by hope.

1.14 Techniques Used

Technique	Methods used	Sample Images
	• Original Picture	
Adding Earthquake Effect	• Resizing the earthquake image: The earthquake image is resized to fit proportionally into the base scene without distorting its appearance, ensuring it integrates well with the composition. • Pasting the earthquake image	

Mirror the crack on the building	• Enhancing Visual Impact: The crack image is enlarged to ensure it makes a strong visual statement in the composition, simulating significant structural damage. • Creating Symmetry: A mirrored version of the crack is generated, providing balance and symmetry to the cracks in the scene. • Blending Seamlessly: Both cracks are strategically positioned to appear natural and blend into the scene, using transparency to integrate the effect without harsh edges.	

Blending Fog into the Composition	• Matching the Scene Dimensions: The fog image is resized to fully cover the entire scene, ensuring uniform application. • Subtle Transparency Adjustment: The fog's opacity is reduced to create a semi-transparent layer that enhances the atmosphere without overpowering the other visual elements. • Natural Integration: The fog is blended with the base scene and effects, adding a realistic atmospheric touch. The semi-transparent fog enhances the dramatic and moody tone of the image.	
Adding Fujiman Figures	• Removing the Fog: The fog layer is excluded, keeping the scene clear and focused on the main elements. • Preparing the Figures: Two Fujiman images (original and mirrored) are resized to fit the scene, ensuring they are prominent yet balanced. • Positioning the Figures: The original is placed on the left, and the mirrored version on the right, creating symmetry and visual balance. • Blending into the Scene: Both figures are integrated naturally, preserving transparency for a dynamic effect.	

Adding CW1, CW2, CW3 and Truck image	• Background Removal: The rembg library removes backgrounds from cw1.jpg, cw2.jpg, and truck.png, isolating subjects for integration. • Positioning CW1 and CW2: CW1 is resized and placed at (550, 900) and (300, 1200), while CW2 is positioned at (1450, 1000). • Placing the Truck: The truck is resized and placed at (950, 1000) after background removal. • Final Layering: These elements are layered over the earthquake, cracks, and Fujiman effects to complete the composition.	

1.15 Screenshot solution

Remove Background

```
from rembg import remove
from PIL import Image, ImageOps
import io

def remove_bg(image_path):
    with open(image_path, "rb") as img_file:
        input_image = img_file.read()
    output_image = remove(input_image)
    return Image.open(io.BytesIO(output_image)).convert("RGBA")

def resize_image(image, size):
    return image.resize(size, Image.Resampling.LANCZOS)

cw1_image_no_bg = remove_bg("cw1.jpg")
cw2_image_no_bg = remove_bg("cw2.jpg")
truck_image_no_bg = remove_bg("truck.png")
```

This program in Python uses the rembg package and Pillow to resize photos and remove their backgrounds. The functions remove_bg and resize_image are defined. After reading an image file from the given location and using rembg to remove its background, the remove_bg method returns the RGBA formatted image. The resize_image function uses the high-quality Lanczos resampling technique to resize an input image to a given size. After that, the script resizes the three images (truck.png, cw1.jpg, and cw2.jpg) to predetermined sizes (100x200, 200x300, and 500x300 pixels, respectively) and removes their backgrounds. For later use, the processed photos are saved in variables called cw1_resized, cw2_resized, and truck_resized.

Mirror the Crack

```
crack_image = Image.open("crack.png").convert("RGBA")
crack_image_left = crack_image.resize((1500, 1500))
crack_position_left = (-200, -550)

base_with_cracks = base_with_earthquake.copy()
base_with_cracks.paste(crack_image_left, crack_position_left, crack_image_left)

crack_image_right = ImageOps.mirror(crack_image_left)
crack_position_right = (base_with_cracks.size[0] - 1200, -300)

base_with_cracks.paste(crack_image_right, crack_position_right, crack_image_right)
```

This Python technique overlays a base image (base_with_earthquake) with mirrored "crack" images to simulate an earthquake. It initially opens a "crack" image (crack.png) and converts it to RGBA format in order to handle transparency. The left-side crack picture is enlarged to 1500x1500 pixels and positioned at (-200, -550) on a duplicate of the base image (base_with_earthquake). Using ImageOps.mirror, the left crack is reflected to create a right-side crack. The mirrored crack is then positioned at (base_with_cracks.size[0] - 1200, -300) with respect to the base image dimensions. Both fractures are glued to the underlying picture while preserving their alpha channels for realistic layering. The final composite image is stored in base_with_cracks.

Conclusion

In conclusion, this project allows for a unique opportunity to study the influence of digital image processing and special effects in re-working urban landscapes—Tokyo in his case—turning mundane images into vivid futuristic, post-apocalyptic, and dystopian visions. Combining various creative techniques such as color grading, atmospheric effects, object overlay, and manipulation of light and shadow, using several Python libraries—including OpenCV and Pillow—we visually dictated this development of Tokyo, from an affluent futuristic city through a devastated wasteland, ultimately to a city evolving towards rebirth.

References

none

Workload Matrix

	Brittney Lau Ko Xuan (TP065461)	Chua Jun Yi (TP065882)	Cheah Mei Suen (TP064568)	Angelina Leanore (TP072929)	Chang Zheng Fang (TP066899)
Introduction	20%	20%	20%	20%	20%
Scene 1	100%				
Scene 2		100%			
Scene 3			100%		
Scene 4				100%	
Scene 5					100%
Conclusion	20%	20%	20%	20%	20%
Signature