SkillsClass Report

RC11
Ruinan Xi
SN:21187542

Contents

Part 1
Concept Explanation

Part 2
Data Vectorization and Computational Analysis

Part 3
Data Visualization and Artistic Expression

Part 1
CONCEPT EXPLANATION

1. Historical development of computation

1.1 Early computation (3000 BCE to 1800 CE):

During the early era of computation, which spans from around 3000 BCE to 1800 CE, people used ancient calculation tools like the abacus, astrolabe, and slide rule. These tools were mainly employed for performing arithmetic and astronomical calculations.

1.2 Mechanical computation (1800 to 1940):

The mechanical computation era, which took place from 1800 to 1940, was characterized by the creation of machines like the difference engine and the analytical engine, invented by Charles Babbage. These mechanical calculators automated complicated computations and served as the basis for the development of modern computing technology.

1.3 Electronic computation (1940 to 1980):

The electronic computation era, spanning from 1940 to 1980, began with the invention of the electronic computer, which included the first programmable digital computer, the ENIAC. This marked a significant revolution in computation, as electronic computers enabled the processing of complex calculations at previously unimaginable speeds.

1.4 Networked computing (1990 to present):

The networked computing era, which began in the 1990s and continues to this day, saw the emergence of the internet, connecting computers across the globe and introducing novel forms of communication and collaboration. The internet also paved the way for the development of cloud computing, big data analytics, and artificial intelligence, significantly advancing the computing industry.

1.5 The connection between ancient and modern computing:

The principles of computation have remained largely unchanged over time, connecting ancient calculation methods with modern computing. While new methods and tools have been developed to improve speed and accuracy, the basic principles of numerical representation and manipulation have persisted. Modern computing systems still rely on these foundational principles, which were established in ancient times.

2. Algorithms, computable functions, Halting Problem

2.1 Algorithms:

An algorithm is a set of mathematical steps and instructions that are used to solve a mathematical problem in a precise and reliable way. It involves using specific numbers and letters to perform calculations and make decisions. By following each step in the correct order, an algorithm ensures that the final result is accurate and correct.

2.2 Computable functions:

This concept refers to the idea that a function is considered computable if there exists an algorithm capable of performing the task that the function requires. Algorithms can be applied to a specific domain in this case. Additionally, these functions can be used without requiring reference to Turing machines.

There are three types of problems that can be identified: decision problems, function problems, and optimization problems. Decision problems are straightforward and simply require determining whether or not a solution exists. Function problems involve identifying the solution required to solve a given problem. Optimization problems focus on identifying the best possible solution for the problem at hand.

For example, the code below is used to determine if the number entered is an even number:

```
def is_even(n):
if n % 2 == 0:
     return True
 else:
     return False

print(is_even(4))
print(is_even(5))
```

In this code, the 'is_even()' function takes an integer 'n' as an input and returns 'True' if the integer is even, and 'False' otherwise.' The function checks if the integer is divisible by 2, and if so, returns 'True.' Otherwise, it returns 'False.'

2.3 Halting Problem:

The halting problem is a crucial issue in the field of computing that aims to determine whether a program will eventually stop running and return an answer or if it will run indefinitely without ever terminating. Essentially, the halting problem seeks to determine if a program will accept or reject certain inputs and when it will terminate. This is a decision-making problem that aims to determine whether a program will run indefinitely, or if it will eventually halt and return an output.

3.(General) recursion, Turing machines, lambda-calculus

3.1 (General) recursion:

Recursion is a vital concept in the field of computability theory that involves defining an object in terms of itself. It typically involves breaking down a complex problem into smaller, simpler subproblems that can be solved by utilizing the same function or method. Recursive functions and algorithms play a significant role in the theory of computability, with many algorithms and functions being defined recursively.

In a recursive function or algorithm, there are two key components: the base case and the recursive step. The base case specifies a condition under which the function or algorithm stops recurring, while the recursive step defines the operation that is repeatedly performed on the input until the base case is reached.

An example of a recursive function in Python that calculates the nth Fibonacci number is as below:

```
def fibonacci(n):
  if n <= 1:  # Base case
     return n
  else:  # Recursive case
     return fibonacci(n-1) + fibonacci(n-2)
```

In this code, the 'fibonacci()' function takes a non-negative integer 'n' as input and recursively calculates the nth Fibonacci number. The base case occurs when 'n' reaches 0 or 1, at which point the function returns 'n'. The recursive case involves calculating the sum of the previous two Fibonacci numbers, which is achieved by calling the 'fibonacci()' function again with 'n-1' and 'n-2' as inputs.

The above function works when the input is 6:

```
fibonacci(6)
= fibonacci(5) + fibonacci(4)
= fibonacci(4) + fibonacci(3) + fibonacci(3) + fibonacci(2)
= fibonacci(3) + fibonacci(2) + fibonacci(2) + fibonacci(1) + fibonacci(2) +
fibonacci(1) + fibonacci(1) + fibonacci(0)
= fibonacci(2) + fibonacci(1) + fibonacci(1) + fibonacci(0) + fibonacci(1) +
fibonacci(0) + 1 + 0
= 8
```

The function calls itself repeatedly with decreasing values of n, until it reaches the base case of n=0 or n=1.

3.2 Turing Machines:

A Turing machine, which was first proposed by Alan Turing in 1936, is a theoretical model used in computer science to represent computation. It is a basic and abstract framework that allows us to define an algorithm, and serves as a foundation for understanding the boundaries of computation.

1. Finite State Machine:
The internal configurations of a Turing machine are represented by a finite set of states, which the machine transitions between based on input symbols and pre-defined rules.

2. Deterministic Finite Automaton:
A Turing machine is considered deterministic if its transition function provides a unique determination of the next state, the symbol to be written, and the direction to move the tape head, for any given input symbol and state.

3. Infinite Tape:
Imagine a large sheet of paper that has been divided into squares, with each square able to contain a symbol such as '1', '0', or a blank space. While it is not actually infinite, this paper can serve as a basic physical representation of an infinite tape.

4. Encoding Turing Machines:
Turing machines can be converted into strings or numbers by encoding their symbols, states, and transition rules. By doing so, it becomes possible to manipulate and analyze Turing machines using other Turing machines.

5. Universal Turing Machine:
An example of a Universal Turing Machine (UTM) is a computer that can execute any program, with the program's code provided as input. In this scenario, the computer serves as a UTM by interpreting and running the program, simulating the actions of the original Turing machine (which in this case is the program) encoded in the input.

6. Turing Completeness:
A computational model or programming language such as Python.

3.3 Lambda-calculus

Lambda-calculus is a fundamental concept in functional programming, and offers an effective approach to analyze computation that complements the study of computability using Turing machines. The lambda-calculus is comprised of three primary elements:

1. Variables: Symbols that represent values, such as x or y.

2. Abstraction: The process of defining a function using lambda notation (λ x.T) involves using the lambda symbol " λ ", followed by a variable (x) and a period, and then the body of the function (T). The variable represents the input to the function, while the body describes the computation or transformation that is applied to the input.

3. Application: Function application is the process of utilizing a function (F) with an argument (M), represented as (F M). In this process, the argument is substituted for the variable in the body of the function to execute the computation.

The following python code is an example using Lambda-calculus:

```
multiply_by_5 = lambda x: x * 5
result = multiply_by_5(5)

print(result) # Output: 25
```

4.Time complexity, Complexity vs Computability

Complexity theory is concerned with analyzing the computational resources required by algorithms to solve problems, such as time and space complexity, and how they scale with input size. This allows us to determine which problems can be efficiently solved in polynomial time and which ones are inherently difficult and require exponential time or more. On the other hand, Computability theory aims to understand the fundamental boundaries of computation by exploring which problems can and cannot be solved by algorithms and the degree to which computation can be automated. This involves analyzing the strengths and limitations of various models of computation, such as Turing machines and lambda calculus. Together, these two theories provide a comprehensive understanding of the nature and limitations of computation, and have practical applications in fields like computer science, mathematics, and engineering.
The following are examples of codes with different time complexity.

```
my_list = [1, 2, 3, 4, 5]
```

Function 1: Linear time complexity, O(n): Sums the values of a list of numbers using a for loop.

```
def sum_list_1(numbers):
      result = 0 # Initialize the result to zero
      for num in numbers: # Loop through each element in the list
            result += num # Add the current element to the result
      return result

# Using function 1 with linear time complexity
print(sum_list_1(my_list)) # Output: 15
```

Function 2: Quadratic time complexity, O(n^2): Sums the values of a list of numbers using nested for loops.

```
def sum_list_2(numbers):
      result = 0 # Initialize the result to zero
      for i in range(len(numbers)):
      # Loop through each element in the list
            for j in range(i+1):
            # Loop through all elements up to the current index
                  result += numbers[j]
                  # Add each element to the result
      return result

# Using function 2 with quadratic time complexity
print(sum_list_2(my_list)) # Output: 15
```

In this example, '`sum_list_1`' has a time complexity of O(n), as it loops through each element in the list once to add its value to the result. On the other hand, '`sum_list_2`' has a time complexity of O(n^2), as it uses nested for loops to loop through each element in the list multiple times, resulting in a slower algorithm for larger input sizes.

For small input sizes, the difference in performance between these two functions may not be noticeable. However, for larger input sizes, the difference in time complexity becomes more significant, with '`sum_list_1`' being much faster than '`sum_list_2`'. It's important to consider the time complexity of an algorithm when designing and implementing programs to ensure efficient execution, especially for large datasets.

5.Measuring complexity, Asymptotic (‘big Oh’) notation, Function growth

5.1 Measure complexity

Complexity measurement quantifies the resources needed to tackle a computational problem, typically focusing on time and space - the time taken and memory used by an algorithm. The prevalent complexity measure is the "Big O notation" providing an estimate on resource usage increase as input size grows. For instance, a time complexity of O(n^2) indicates a quadratic growth in time with input size. Other complexity types include "space complexity" for memory usage and "communication complexity" for information exchange within a system.

5.2 Asymptotic (‘big Oh’) notation

The 'Big O' notation, represented as O(f(n)), provides an estimate of how the performance of a function escalates as the input size, denoted by "n", increases. For instance, when we express an algorithm's time complexity as O(n), it signifies that the algorithm's execution time increases at a rate directly proportional to the size of the input.
Here are two simple Python functions to illustrate the concept of Big O notation:

Function 1: O(n) time complexity:

```
def linear_time(n):
      for i in range(n):
                  print(i)
```

Function 2: O(n^2) time complexity:

```
def quadratic_time(n):
      for i in range(n):
                  for j in range(n):
                        print(i, j)
```

The function '`linear_time`' exhibits a time complexity of O(n) since it executes a single action - printing a number - for every item in the list. Therefore, if the list size doubles, the execution time is likely to double as well. On the other hand, the function '`quadratic_time`' shows a time complexity of O(n^2) as it operates on each combination of two elements in the list. This means that if the list size doubles, the execution time is expected to quadruple (as 2^2 equals 4).

In practice, an algorithm's time complexity could vary depending on various elements such as the specific operations involved, the utilized data structures, and the nature of the input data.

5.3 Function growth

Function growth pertains to how a function's demand for time or space evolves in relation to the increase in its input size. The growth rate is usually articulated using the big-Oh notation, serving as a maximum limit for how quickly the function's resource requirements could expand given the size of its input.

6.Algorithm design and optimisation, Divide-and-conquer paradigm

6.1 Algorithm design and optimisation

The subjects of algorithm creation and enhancement are crucial in the field of computer science, as they contribute to the refinement and performance improvement of computer applications.

Algorithm design involves devising a sequential plan to address a certain issue. This plan usually accepts input data, manipulates it, and generates output data. The objective is to formulate an algorithm that is accurate, efficient, and straightforward to comprehend and execute. Optimization is the process of enhancing an algorithm's efficiency by minimizing the time or space it requires to accomplish a task. Various strategies are employed in optimization, including time complexity analysis, space complexity analysis, the divide and conquer approach, dynamic programming, and greedy algorithms.

Divide-and-conquer is a widely used approach in algorithm design that breaks down a complex problem into smaller, more manageable subproblems. These subproblems are solved individually, and their solutions are then brought together to address the main problem. The divide-and-conquer technique is characterized by three main phases:

6.2 Divide-and-conquer paradigm

1.Divide: Divide the problem into a number of subproblems that are smaller instances of the same problem.

2.Conquer: Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

3.Combine: Combine the solutions to the subproblems into the solution for the original problem.

The following is an example of binary search in Python:

```
def binary_search(arr, low, high, x):
      if high >= low:
            mid = (high + low) // 2

            if arr[mid] == x:
                  return mid
            elif arr[mid] > x:
                  return binary_search(arr, low, mid - 1, x)
            else:
                  return binary_search(arr, mid + 1, high, x)
      else:
            # Element is not present in array
            return -1
```

In this algorithm, the array is split into two halves during the divide phase. The conquer phase involves recursive function calls on one of these halves. The combine phase involves no additional work as the solutions to the subproblems, namely the element's location in the subarray, directly provide the solution to the initial problem.

7.Space complexity, Complexity classes

7.1 Space complexity

Space complexity quantifies the memory usage of an algorithm relative to the size of its input. This enables us to gauge how the algorithm's memory demand scales with larger input sizes, thereby assisting us in selecting the most memory-efficient algorithm for a particular task.

7.2 Complexity classes

Complexity classes, in the context of computational complexity theory, are groupings used to categorize problems according to their resource demands for resolution. The resources in question may involve the duration (time complexity) or memory utilized (space complexity) by a deterministic or non-deterministic Turing machine. Here are some examples of prevalent complexity classes:

P: This category comprises decision problems that a deterministic Turing machine can resolve within polynomial time.

NP (Nondeterministic Polynomial time): This category includes decision problems whose solutions can be confirmed to be correct in polynomial time using a deterministic Turing machine. Essentially, if you have a "yes" solution to a problem, you can verify its accuracy within polynomial time.

NP-Complete: These represent the most difficult problems within the NP class. The key characteristic of NP-complete problems is that if any one of them can be resolved within polynomial time, then all problems in the NP class can also be tackled within polynomial time.

NP-Hard: These problems are "at least as hard as" the hardest problems in NP. If a problem is NP-hard, it doesn't necessarily have to be in NP, and it doesn't have to be a decision problem.

EXPTIME: This category comprises decision problems that can be resolved within exponential time using a deterministic Turing machine.

Complexity classes play a significant role in aiding computer scientists to comprehend the intrinsic difficulty of problems and evaluate the efficacy of algorithms in resolving them. It's worth mentioning that the P versus NP problem, listed among the seven "Millennium Prize Problems," is yet to be solved. This problem essentially queries whether every problem, whose solution can be efficiently verified, can also be efficiently solved.

We can write Python code to illustrate different complexities (time complexities specifically) that often appear in complexity classes. Examples are as below:

O(1) - Constant Time Complexity:

```
def constant_complexity(n):
      return n[0]
```

O(n) - Linear Time Complexity:

```
def linear_complexity(n):
      for i in n:
            print(i)
```

O(n^2) - Quadratic Time Complexity:

```
def quadratic_complexity(n):
      for i in n:
            for j in n:
                  print(i, j)
```

O(log n) - Logarithmic Time Complexity:

```
def logarithmic_complexity(n):
      i = 1
      while i < n:
            i *= 2
```

8.Sorting algorithms, Matrices, Matrix multiplication

8.1 Sorting Algorithms

Sorting algorithms are methods that organize a collection of data into a particular sequence. These algorithms play a crucial role in data analysis and management. Various kinds of sorting algorithms exist, such as insertion sort, selection sort, merge sort, quicksort, and bubble sort, among others.

Here's an example of python code:

```
def bubble_sort(numbers):
      for i in range(len(numbers)):
            for j in range(0, len(numbers) - i - 1):

            if numbers[j] > numbers[j+1] :
                  numbers[j], numbers[j+1] = numbers[j+1], numbers[j]
      return numbers

numbers = [44, 35, 26, 12, 27, 18, 95]
print(bubble_sort(numbers))
```

This script establishes a function, named 'bubble_sort', that organizes a list of numbers. It achieves this by continuously traversing the list and swapping neighboring numbers that aren't in the correct sequence. This procedure is iterated until the whole list is arranged orderly. The function is subsequently put to the test on a list of numbers, and it successfully arranges the list in ascending order.

8.2 Matrix and Matrix Multiplication

A matrix is a numerical grid organized into columns and rows. In Python, a matrix can be represented using a 2D array; for instance,M = . Furthermore, using the NumPy library, a matrix can also be represented as M = np.array([[1, 2], [3, 4]]). The following are the steps for matrix multiplication of matrices A and B:

```
import numpy as np
M = np.array([[1, 2], [3, 4]])

print(M)
```

1.Compatibility Requirement: The multiplication operation between two matrices, A and B, can only proceed if the number of columns in matrix A is equal to the number of rows in matrix B.

2.Resultant Matrix Dimensions: The resulting matrix, C, will have dimensions that correspond to the number of rows from matrix A and the number of columns from matrix B.

3.Computing Elements in Matrix C: Each element in the resulting matrix C is calculated following a specific formula or rule.

9.Optimizing time complexity

In our design projects, we sometimes need to retrieve specific movie titles from our movie library. We have therefore written a function code to manipulate it. The python code is as below:

```
def find_film_mane(filmList, targetName):
      for i in range(len(filmList)):
            if filmList[i] == targetName:
                  return i
      return False
```

We analyze that the time complexity of this function is O(n), where n is the length of the list, because in the worst-case scenario, it must iterate through every element in the list. Now, suppose we know that the list is sorted. We can optimize it by using a binary search instead of a linear search to find the target value:

```
def find_film_name_opt(filmList, targetName):
      low, high = 0, len(filmList) - 1

      while low <= high:
            mid = (low + high) // 2
            if filmList[mid] == targetName:
                  return mid
            elif filmList[mid] < targetName:
                  low = mid + 1
            else:
                  high = mid - 1
      return False
```

The time complexity of this function is O(log n), where n is the length of the list, because with each iteration, it halves the size of the list segment it is looking at. This is a significant improvement over the original function when the list is large.

Another example is that, when we want to find duplicate films in our film library, we want to compare each film with every other one. So we write python code as below:

```
def has_duplicate_film(filmList):
      n = len(filmList)
      for i in range(n):
            for j in range(i + 1, n):
                  if filmList[i] == filmList[j]:
                        return True
      return False
```

This code runs in O(n^2) time complexity because, for each element in the list, it checks equality with every other element. We can optimize this by using a set in Python, which has an average time complexity of O(1) for checking if an element is in the set. The optimized code is as below:

```
def has_duplicate_film(filmList):
      seen = set()
      for num in filmList:
            if num in seen:
                  return True
            seen.add(num)
      return False
```

This optimized code runs in O(n) time complexity. It goes through the list only once and uses the set to check and remember elements it has seen before. As a result, it is significantly faster than the previous version for large lists.

Part 2
DATA VECTORISATION & COMPUTATIONAL ANALYSIS

1. Vectorise images and film

1.1 Neural Network

1.1.1 Concept of Neural Network

A neural network is a machine learning model designed to mimic the human brain's architecture and functionality, which involves a network of interlinked neurons cooperating to solve intricate problems. A basic neural network comprises three components:

The input layer, which receives the data.
The hidden layer, consisting of numerous layers that learn and derive features from the input data via a series of transformations.
The output layer, which delivers the expected output or forecast.

In a prevalent kind of neural network known as a feedforward network, data flows unidirectionally from the input to the output layer. Every layer gets input from its preceding layer and generates output for the subsequent layer. The final output is given by the last layer, while the penultimate layer generates intermediate representations assisting the network in making its ultimate prediction.

1.1.2 The difference between the last and second-to-last layer

The final, or output layer generates the ultimate predictions. Its activation function and neuron count may change based on the problem being tackled. Pre-final Layer, also known as a dense or fully connected layer, this layer resides just before the output layer. It amalgamates features learned by prior layers, including convolutional and pooling layers, into more abstract representations. It helps the neural network recognize complex patterns in the data, enabling the final layer to make precise predictions or classifications.
In essence, the primary distinction between these two layers lies in their roles. The pre-final layer extracts high-level features, while the final layer leverages these features for final predictions or classifications.

1.2 Vectorisation of image and video

1.2.1 Image Recognition Model

Convolutional Neural Networks (CNNs) are often utilized for Image Classification tasks. The procedure generally involves the following steps: Initially, we extract samples from an image by selecting pixel blocks and repeating this process for each block (equivalent to each layer in the neural network) until we reach the output layer. Each sample (or neuron in the preceding layer to the output layer) assigns a weight to the final output, representing the probability of each class label. The sum of all output values equals one, and the class with the highest value represents the predicted category.

We import the MobileNet model from Keras, which is a widely used pre-trained model for image categorization tasks. The model initialization utilizes the following settings:

```
model = tf.keras.applications.mobilenet.MobileNet(
#The 3 is the three dimensions of the input: r,g,b.
input_shape=(224, 224, 3),
include_top-False,
pooling-'avg
)
```

We get our own image library from Google, and classify them manually in different files. Then we can load them in and batch train the model.

```
batch_size = 32
history = model.fit(
      x=training,
      steps_per_epoch=training.samples // batch_size,
      epochs=1,
      validation_data=validation,
      validation_steps=validation.samples // batch_size
```

Then we allow the model to predict the extent the images are similar to each classification.

```
            fox_path = '/content/drive/MyDrive/RC11-SkillsClass2022/images/red-
fox-in-winter.jpg'
            fox = load_image(fox_path)
            tf.keras.preprocessing.image.load_img(fox_path, target_
size=(224,224))
            print(f'''fox:{np.round(model.predict(fox),2)}''')
```

The outcome is as below:

```
1/1 [==============================] - 0s 146ms/step
fox:[[0.72 0.28]]
```

1.2.2 Video Classification

The PyTorch-based 3D ResNet model for video classification, trained on 400 action classes, can provide class labels and scores for each 16-frame segment, indicating potential activities. It can also output 512-dimensional feature vectors in feature mode. Insights about the number of clips, labels, and scores are gathered from the video's JSON file. Sorting these labels and scores allows for the prediction of the most likely activities in the video clips.

There are two modes that can be used in the model, score mode and feature mode. In score mode, the model delivers labels and scores of video clips, aiding in video segmentation based on different activities. Feature mode, on the other hand, allows for similarity searches among video segments, making it possible to locate similar clips within a video library. The initial step involves merging all .json files in the output folder into one comprehensive dictionary.

The process starts by extracting the features of the key film's clips from the feature dictionary, storing them in a list named 'listkeyfeatures'. The function then iterates through the remaining films in the feature dictionary, collecting their clip features into 'filmfeatures'. For each clip in every film, the function calculates the distance between the clip's features and those of the key film's clips. It then computes the average distance across all clip features in the two films. If the computed distance is smaller than the current best matches, the function updates the match list with these new matches. Finally, the function returns a list of fragments sorted by their distance, with the closest matches at the top of the list.

The function used to find the best match clips is as below:

```
def searchForMatch(keyFilmName, featureDictionary, frame = None, fps = 26,
nBestMatches = 1):
    fragments = []
    keyClips = featureDictionary[keyFilmName]['clips']
    keyFeatures = []
    for c in keyClips:
        keyFeatures.append(np.array(c['features']))

    if frame:
        ft = 16/fps
        sf = int(frame[0]/ft)
        ef = int(min(frame[1]/ft, len(keyFeatures)-1))
        keyFeatures = keyFeatures[sf:ef]

    for film in featureDictionary.keys():
        if not film == keyFilmName:
            filmClips = featureDictionary[film]['clips']
            filmFeatures = []
            for c in filmClips:
                filmFeatures.append(np.array(c['features']))
            for i in range(len(filmFeatures)-len(keyFeatures)):
                distance = 0
                for j in range(len(keyFeatures)):
                    d = np.linalg.norm(filmFeatures[i+j]-keyFeatures[j])
                    distance += d
                distance = distance/len(keyFeatures)

                a = fragmentsExists(fragments, [film, i, len(keyFeatures),
distance])
                if a:
                    fragments = a
                elif len(fragments) < nBestMatches:
                    fragments.append([film, i, len(keyFeatures), distance])
                    fragments.sort(key = lambda x : x[-1])
                elif distance < fragments[-1][-1]:
                    fragments.pop()
                    fragments.append([film, i, len(keyFeatures), distance])
                    fragments.sort(key = lambda x : x[-1])
```

1.2.3 Application of image classification models in design project

Our design starts from some personally recorded videos and film clips that have scenes in Istanbul. We search for similar video clips in the film library based on the features of these clips. Therefore, we use an image classification model. Firstly, we extract key frames from the input segment and compare them with the frames extracted from the videos in the film library. Eventually, we find the best match frame and thus determine the position of the desired segment.

Here is part of the Python code we use:

```
def findFramesByImage(filmFeatures, imagePath, model):
      imF = processImage(imagePath, model)

      bestFrameDist = 100000000
      bestFrameFilm = None
      bestFrameSecond = 0

      for f in filmFeatures:
            dist = np.linalg.norm(f['features']-imF)
            if dist < bestFrameDist:
                  bestFrameDist = dist
                  bestFrameFilm = f['film']
                  bestFrameSecond = f['second']

      return [bestFrameFilm, bestFrameSecond]
```

2.Design of algorithms to discern structures

2.2.1 Nearest Neighbor-MST Diagram

To illustrate the similarity between images, we use the nearest neighbor module, which seeks images that bear a close resemblance to the original. We then employ NetworkX to map these images based on their likeness. This process helps us to construct a tree structure. Unlike the SOM approach, this tree diagram visualization emphasizes the distance between two images, which symbolizes their degree of similarity in the feature space.

The procedure begins by converting the images into vectors to extract digital data, determine the location of each image, calculate its edges to its neighbor, and then establish a connection. The following steps elaborate on this process in detail:

1.The pre-trained MobileNet model is loaded for vectorization.

2.The mask files are inputted, and 150 images of the masked objects from Segment are loaded.

3.The MobileNet model is applied to the images using the predict method, which outputs the features as an array of vectors.

The code is as below:

```
def processImage(imagePath, model):
      im = load_image(imagePath)
      f = model.predict(im) [0]

features = []
def masks_list(masks,folder):
      for m in masks:
      path = os.path.join(folder, m)
      f = processImage(path, model)
      features.append(f)
return features
masks_list(masks, '/content/drive/MyDrive/for_som_2')
```

The process begins by identifying the nearest neighbors and displaying the images. With 'n_neighbors' set to 3, the two images most closely related in terms of features to each image are identified, and the distances between them are calculated. The image numbers are then displayed by outputting the indices. An example from one of the lists is showcased by extracting the number of each image to retrieve it from its file folder.

```
nbrs = NearestNeighbors(n_neighbors = 3, algorithm='auto').fit(features)
distances, indices = nbrs.kneighbors(features)
indices

      def showNearestNeighbours (index, indices, files, path):
            for i in indices [index]:
            f = files [i]
            print(f)
            display (iImage (filename = os.path.join (path, f)))
      showNearestNeighbours(100, indices, masks, '/content/drive/for_
som_2')
```

The creation of edges between images is initiated by invoking the NetworkX module to construct a new Graph and appending the images as nodes to the graph. Following this, edges (or relative indexes) are created for every pair of images. A 'for' loop is utilized to retrieve each index from the indices list and extract its corresponding distance from the distances list. The first image in each index is then designated as the node. Subsequently, an 'edges' list is generated to accumulate images in each index, excluding the first one. Finally, another 'for' loop is used to sequentially export the distances between the node image and the remaining images.

```
def createEdges (indices, files, distances):
      weightedEdges = []
      for i in range (len(indices)):
      index = indices[i]
      dist = distances[i][1:]

      node = files [index [0]]

      edges = []
      for e in index [1:]:
            edges.append (files[e])

      for i in range (len(edges)):
            weightedEdges.append ((node, edges[i], dist[i]))

edges = createEdges(indices, masks, distances)
```

Begin by incorporating the edges into the Graph, which then holds both nodes and relevance data. Visualization of the graph is achieved through invoking the matplotlib.pyplot module. Here, the 'fig' variable retains a reference to the newly created figure, while the 'ax' variable retains a reference to the plot axes. Subsequently, the Graph, complete with labels, is displayed by invoking the NetworkX module.

```
fig, ax = plt.subplots(figsize = (12,12))
nx.draw(G, with_labels=True)
```

The Minimum Spanning Tree (MST) is employed to enhance the efficiency of the Graph. The MST, which is a subset of the edges of the original Graph, connects all vertices while maintaining the least possible total weight. The utilization of the MST simplifies the Graph by reducing its branches, thereby making its structure significantly more coherent.

3.Vectorisation of texts

Vectorisation of text, often known as text encoding or text representation, is the procedure of transforming text data into a numerical form that can be understood and manipulated by machine learning models. Given that the majority of machine learning models require numerical data, it's essential to transform raw text, which is naturally non-numerical, into a numerical format for efficient analysis and modeling. Various techniques are available for vectorising text data, such as: Bag of Words (BoW), TF-IDF (Term Frequency-Inverse Document Frequency), Word Embeddings (like Word2Vec, GloVe, FastText and BERT (Bidirectional Encoder Representations from Transformers).

We use the Gensim library for text vectorisation in our text similarity analysis. This model transforms text into numerical representations for similarity computations. We first convert the text into a corpus (a list of words or a bag-of-words representation), and then use the 'similarity' class to create an index for similarity queries. Once we have numerical representations of sentences, we can measure their similarity. Finally, we sort these scores, which can be used to rank sentences or documents based on their similarity to a query.

Steps:

1.Text prepossessing

In preparation for assessing sentence similarity, text must be preprocessed. This includes transforming to lowercase, removing stop words, and applying stemming or lemmatization. Gensim provides these tools. To analyze a sentence or paragraph, we break the text into individual sentences, and extract every word for computation.

```
for p in combined_paragraphs[30:40]:
      print(p)
      print('-----------------')

processed_corpus = [[token for token in t if frequency [token]>1]
for t in texts]
```

2.Vectorisation

In this step, we employ the Bag of Words (BoW) method, which forms a document matrix where each row is a document and each column is a unique word in the vocabulary. This matrix is populated with the occurrence count of each word in the respective document.

```
from gensim import corpora
dictionary = corpora.Dictionary(processed_corpus)
```

3.Similarity Index

Gensim provides multiple approaches to construct similarity indices using different similarity measures, including cosine similarity. These indices aid in conducting effective similarity searches. The Similarity class, which creates an index for executing similarity queries, is widely used.

```
words = 'clock'.lower().split()
print(tfidf[dictionary.doc2bow(words)])
```

4.Calculating Similarity

In order to ascertain the similarity between two sentences, the sentences are usually transformed into numerical formats like bag-of-words or TF-IDF vectors. Then, a similarity index is employed to calculate the similarity score. Gensim offers functions to convert sentences into these vector forms and to carry out similarity computations.

```
      index_tmpfile = get_tmpfile('index')
      index = Similarity(index_tmpfile, bow_corpus, num_features =
len(dictionary))

      # input the sentence you want to find its similar one
      query_document = 'The Grand Bazaar, one of the largest and oldest markets
in the world'.lower().split()
      query_bow = dictionary.doc2bow(query_document)
      sims = index[query_bow]
      # The sims variable will contain a list of similarity scores. The position
of each score in the list corresponds to the index of the document in the bow_
corpus. You can use these scores to rank the documents based on their similarity
to the query document.

      for document_number, score in sorted(enumerate(sims), key = lambda x:x[1],
reverse = True):
      print(document_number, score)
```

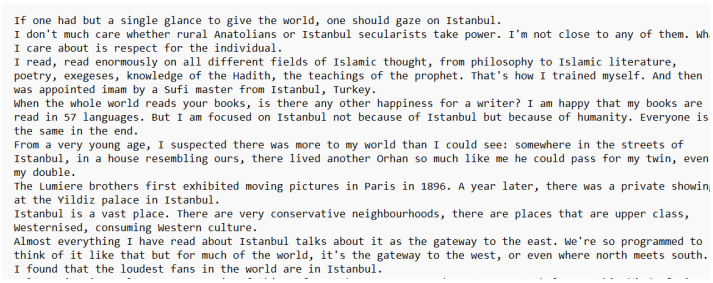
5.Interpreting the Similarity Scores

Similarity scores vary between 0 and 1, with 0 meaning no similarity and 1 indicating that the sentences are identical.

4.SOM text clustering

Text clustering using Self-Organizing Maps (SOM) is a technique for grouping and organizing textual data based on similarity. Here are the general steps involved in text clustering using SOM:

Preprocess the text data:
Start by preprocessing the text data to ensure it is in a suitable format for analysis. This typically involves tasks such as removing punctuation, converting to lowercase, tokenization. Here I have a scraped text file that contains 225 sentences. I want to use SOM to cluster them by similarity.


If one had but a single glance to give the world, one should gaze on Istanbul.
I don't much care whether rural Anatolians or Istanbul secularists take power. I'm not close to any of them. Wh
I care about is respect for the individual.
I read, read enormously on all different fields of Islamic thought, from philosophy to Islamic literature,
poetry, exegeses, knowledge of the Hadith, the teachings of the prophet. That's how I trained myself. And then
was appointed imam by a Sufi master from Istanbul, Turkey.
When the whole world reads your books, is there any other happiness for a writer? I am happy that my books are
read in 57 languages. But I am focused on Istanbul not because of Istanbul but because of humanity. Everyone is
the same in the end.
From a very young age, I suspected there was more to my world than I could see: somewhere in the streets of
Istanbul, in a house resembling ours, there lived another Orhan so much like me he could pass for my twin, even
my double.
The Lumiere brothers first exhibited moving pictures in Paris in 1896. A year later, there was a private showin
at the Yildiz palace in Istanbul.
Istanbul is a vast place. There are very conservative neighbourhoods, there are places that are upper class,
Westernised, consuming Western culture.
Almost everything I have read about Istanbul talks about it as the gateway to the east. We're so programmed to
think of it like that but for much of the world, it's the gateway to the west, or even where north meets south.
I found that the loudest fans in the world are in Istanbul.

```
file = 'C:/Users/Ruinan/Homework/Self-Organizing-Map-SOM-main/output.txt'
with open(file, 'r', encoding='utf-8') as f:
    texts = f.readlines()
len(texts)
output:225
```

Vectorize the text:
Convert each text document into a numerical representation that can be processed by the SOM. Use the vectorizing method mentioned above:
 # Create a dictionary from the corpus of documents

```
dictionary = corpora.Dictionary([nltk.word_tokenize(text.lower()) for text in
texts])
# Convert each document to a bag-of-words representation using the dictionary
bow_corpus = [dictionary.doc2bow(nltk.word_tokenize(text.lower())) for text in
texts]
# Create a TF-IDF model from the bag-of-words corpus
tfidf = models.TfidfModel(bow_corpus)
feature = []
for bow_doc in bow_corpus:
    tfidf_doc = tfidf[bow_doc]
    vec = gensim.matutils.sparse2full(tfidf_doc,len(dictionary))
    vec = np.array(vec)
    feature.append(vec)
The feature output like this: [array([0.03395823, 0.00064878, 0.04751607, ..., 0.
, 0.        ,
        0.        ], dtype=float32),
 array([0.        , 0.00139111, 0.        , ..., 0.        , 0.        ,0.
], dtype=float32),
 array([0.11818164, 0.00096766, 0.02362368, ..., 0.        , 0.        ,0.
], dtype=float32),
 array([0.0194728 , 0.00111609, 0.02724733, ..., 0.        , 0.        ,0.
], dtype=float32),
 array([0.07761609, 0.00037072, 0.05430217, ..., 0.        , 0.        ,0.
], dtype=float32),
 array([0.02479359, 0.00047369, 0.06938488, ..., 0.        , 0.        ,0.
], dtype=float32),
 array([0.0774818 , 0.00098687, 0.03613881, ..., 0.        , 0.        ,0.
], dtype=float32),…]
```

Create a SOM and train:
Initialize a SOM, which is a grid-like structure of nodes or neurons. Each neuron represents a cluster or group that the text documents will be assigned to. Here I use SOMPY to achieve this.

SOMPY, which stands for Self-Organizing Maps for Python, is a Python library that provides a versatile implementation of self-organizing maps (SOMs). SOMs are unsupervised machine learning algorithms used for clustering, visualization, and dimensionality reduction tasks.

```
n_rows, n_cols = 10, 10 # specify the SOM size
som = sompy.SOMFactory.build(np.array(feature), mapsize=[n_rows, n_cols],
initialization='pca')
som.train(n_job=1, verbose='info')
```

```
Training...
pca_linear_initialization took: 0.025000 seconds
Rough training...
radius_ini: 2.000000 , radius_final: 1.000000, trainlen: 14

epoch: 1 ---> elapsed time:  0.018000, quantization error: 40.421006

epoch: 2 ---> elapsed time:  0.019000, quantization error: 40.357387

epoch: 3 ---> elapsed time:  0.017000, quantization error: 39.769189

epoch: 4 ---> elapsed time:  0.018000, quantization error: 39.593519

epoch: 5 ---> elapsed time:  0.017000, quantization error: 39.507461

epoch: 6 ---> elapsed time:  0.018000, quantization error: 39.453175
```

Cluster assignment: Once the SOM training is complete, assign each text document to its corresponding cluster based on the winning neuron. The winning neuron is the neuron with weights most similar to the vectorized representation of the text document.
Here I define a function for find BMU(best matching unit)

```
def find_bmu(som, vec):
    # Get the distances from the input vector to each neuron in the SOM
    distances = np.linalg.norm(som.codebook.matrix - vec, axis=1)
    # Get the index of the neuron with the smallest distance (i.e., the BMU)
    bmu_index = np.argmin(distances)
    # Convert the index to a 2D coordinate (i.e., the BMU's position in the SOM
grid)
    bmu_coord = np.unravel_index(bmu_index, som.codebook.matrix.shape[:2])
    return bmu_coord
```

Visualization: Optionally, visualize the results to gain insights and interpret the clustering outcome. Visualization techniques could include plotting the SOM grid with different colors representing different clusters or using other visualization methods to explore the relationships between clusters. Here I choose to use UMatrixView:

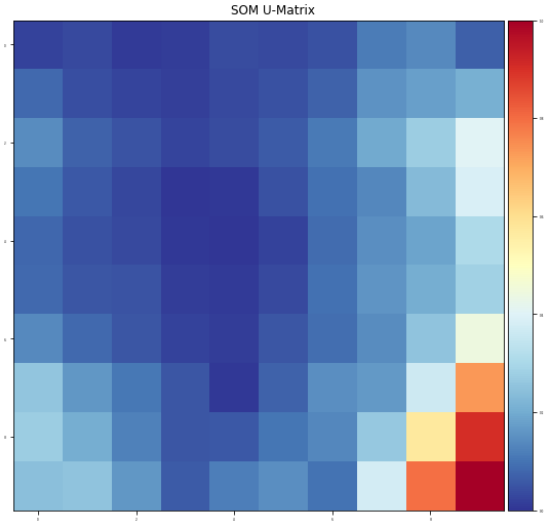
```
from sompy.visualization.umatrix import UMatrixView
umat = UMatrixView(width=20, height=20, title='SOM U-Matrix')

umat.show(som)
```

In the U-matrix visualization, the color of the units represents the level of similarity or dissimilarity between neighboring nodes in the self-organizing map (SOM). The U-matrix color spectrum represents the local density of the data and can be interpreted as clusters or boundaries. Dark blue regions often correspond to clusters or groups of similar data points, while lighter blue and red regions indicate areas where dissimilar or diverse data points reside.

Postprocessing and analysis: After clustering, you can analyze the clusters to gain a deeper understanding of the grouped text documents. This might involve identifying the most representative documents within each cluster, evaluating the cluster quality, or further analyzing the content of the documents within each cluster.

It's worth noting that the specific implementation details and techniques used can vary depending on the programming language or libraries you choose to work with for text preprocessing, vectorization, and SOM training.



4.Represent and visualise data in GODOT

GODOT is a unique and powerful game engine that employs custom programming to facilitate a myriad of operations. It shares many similarities with the Blender software, particularly in the domain of functionalities. However, where GODOT truly shines is in its interaction between the user interface and the end user, establishing a dynamic, responsive environment that proves beneficial in game design and beyond.

To explore the full potential of this interaction, two experiments were undertaken. The aim was to investigate the possibilities of creating a more profound connection between the project and its audience. This exploration might not only enhance the user experience but also forge a new path for future design projects, potentially revolutionizing the way we approach and execute such tasks.

At the core of GODOT's programming principle is a focus on objects and the events associated with them, a philosophy known as object-oriented programming. This approach encapsulates data, or properties, of objects within a JSON file. Once this file is imported into the GODOT engine, it provides a blueprint for objects, bestowing them with a range of properties like names and original coordinates. This method allows for a level of customization and control that is integral to the design and functioning of the interactive environment.

The 3D experiment conducted employed a model that was generated from one of the objects featured in the four-version videos. This component has significant potential implications for future projects. It allows for a tangible, visual representation of negotiation methods, creating a more immersive and understandable experience for users. Moreover, it provides a platform to demonstrate the functionality and adaptability of the design.
By applying different options to this model, users can see first-hand the impact of their choices, making the design process more transparent and participatory. This level of user involvement could be a game-changer in design projects, creating a more engaging and user-friendly environment.

Part 3

DATA VISUALIZATION & ARTISTIC EXPRESSION

1. Scraping and mapping

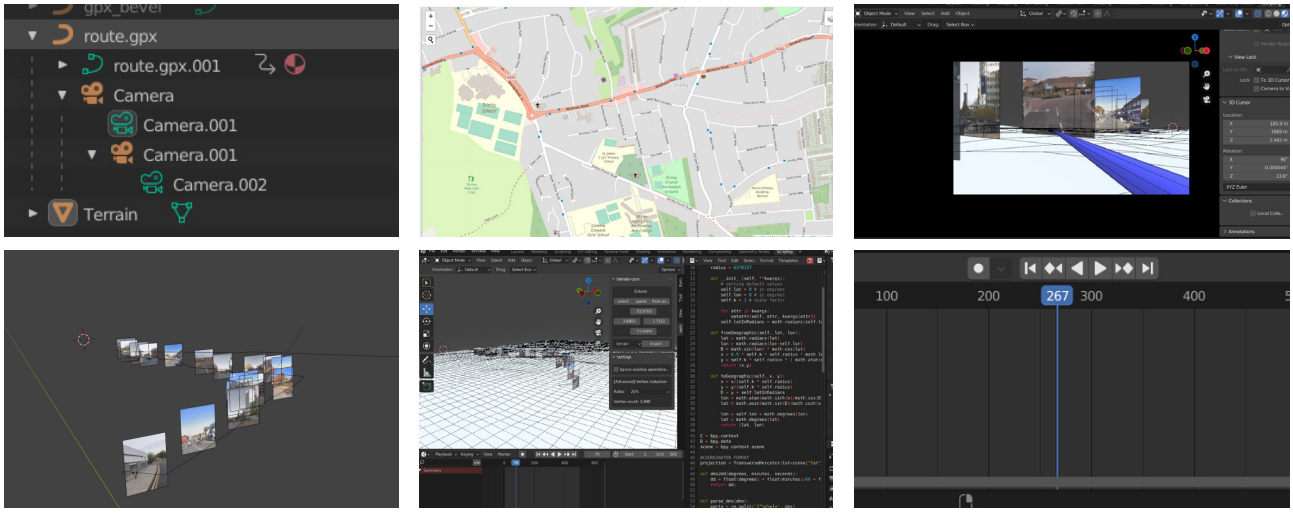
1.1 Create gpx file of walking path
*(**As I can't get access to google maps on my Iphone, I have no way to record my walking paths. So here I have used the files provided by Joris in class.)*

1.2 Retrieve Metadata of Photo
The photos provided by Joris, which were taken along the walking path, contain individual information such as location coordinates, shooting angles, and time, etc. Git Bash was used to extract the location coordinate information of all the photos.

1.3 Import path and photos
Utilize the script window in Blender to bulk import photos into the model based on their location data. Subsequently, incorporate the walking path and position it accurately.

1.4 Animation
Position the camera at the path's beginning. Tweak the camera's angle and focus for optimal results. Then, attach the camera to the path to ensure its movement is guided along the path.

1.5 Import Terrain
To provide context and enhance the entire video, terrain information can be incorporated using the Blender OSM add-on. The desired area can be selected using a rectangular wireframe, from which location information can be extracted. This location information can then be imported into Blender, supplying a realistic terrain backdrop for the video. This process follows the successful creation of the camera path.



2. Mapping the earthquake

First extract frames from the video with a piece of code:

```
ffmpeg -i Video.mp4 -qscale:v 2 -vf fps=3/1 output_%03d.jpg
```

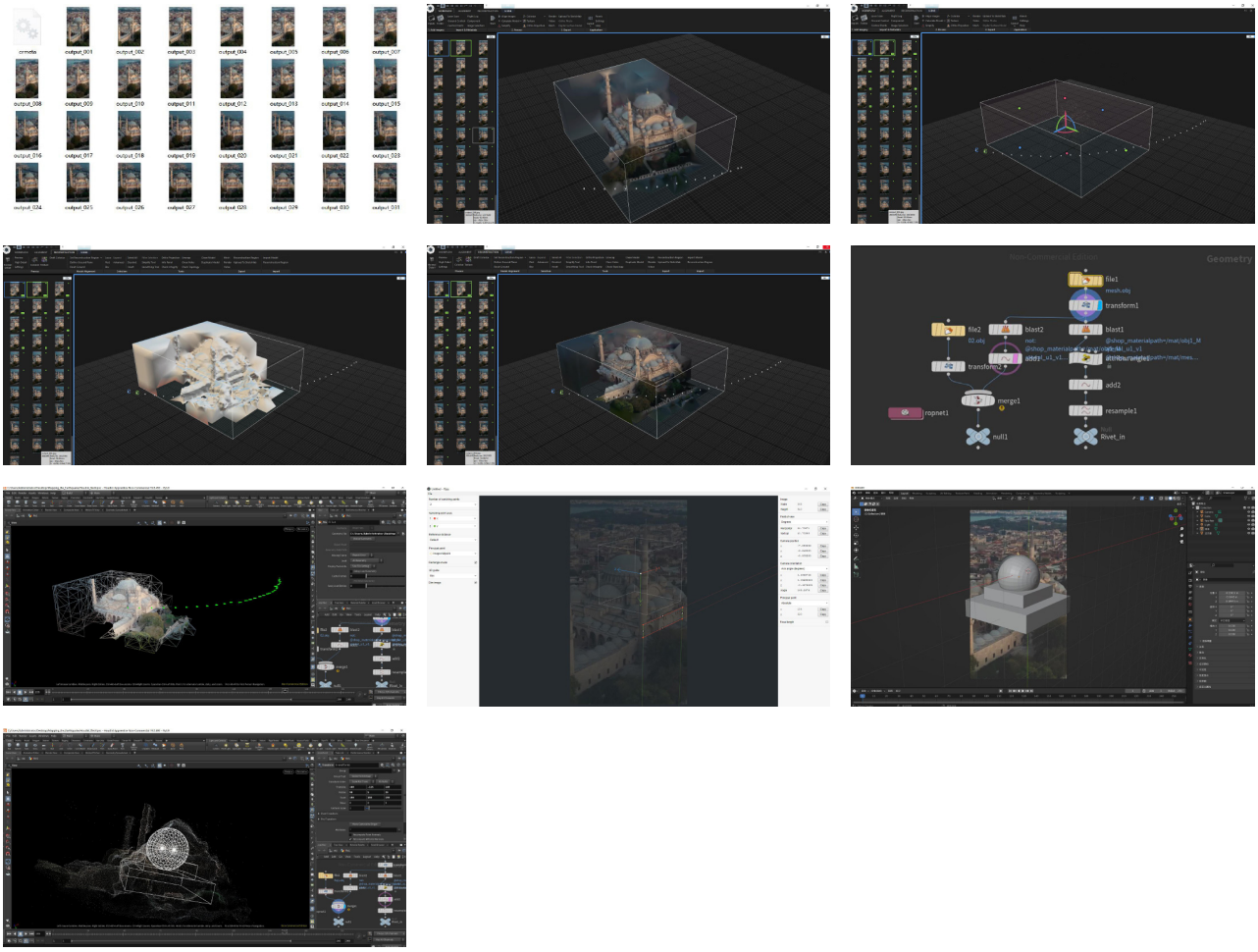
We've chosen a 12-second video for this task, extracting 3 frames per second, which totals 36 frames. The extraction results are as follows:

The subsequent step involves importing these frames into Reality Capture to begin the reconstruction process. The initial phase includes aligning the cameras to ensure their correct positioning. Following this, normal details and textures are constructed, resulting in a comprehensive model accompanied by cameras. Finally, the .obj file is exported.

Once the model is loaded into Houdini, we modify its orientation, location, and scale. We then create camera paths using camera objects and connect the camera to these paths. This procedure yields the reassembled video.

Select a single frame and import it into fSpy, then establish the axes using rectangle mode. Afterward, import the fSpy file into Blender, ensuring the axis direction is correct. Construct the model and finally import it into Houdini.

Switch to the camera view and play the video to replicate the scene and path. Following this, render and export the frames in the .png format.



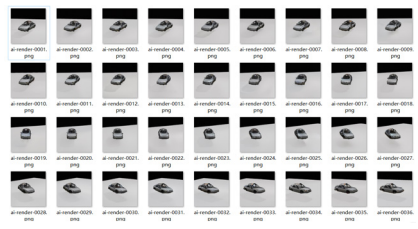
3. Stable diffusion in blender

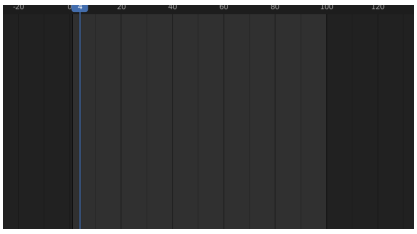
Stable Diffusion is a sophisticated deep learning model specifically engineered for text-to-image functionalities. Its principal role entails generating detailed images in response to text descriptions. Beyond this primary usage, it also exhibits flexibility in handling diverse tasks including inpainting, outpainting, and executing image-to-image translations under the guidance of text prompts. At the inception of this project, I employed a car model that had been scanned earlier, importing its OBJ file along with the associated texture into the Blender platform.

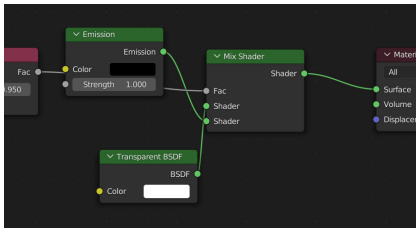
The objective was to showcase the bear model, around which the stable diffusion was implemented, from various angles. This ideally involved the camera path circumnavigating the bear model. To facilitate this, an empty plain axes was set up and paired with the camera, enabling the camera to orbit around the model. Then, in order to animate the camera movement, keyframes were inserted in frames 1 and 100. This setup allowed the camera to move in accordance with these keyframes, providing a predetermined path for the animation sequence. The ultimate aim was to render an animation that effectively displays the model from different perspectives.

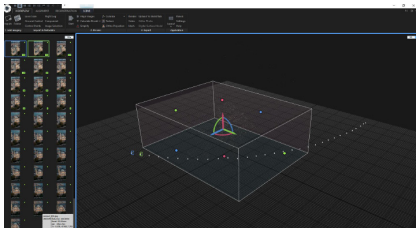
In the AI Render prompt, a more detailed description of the car was provided through additional keywords. The outcome of executing the render command resulted in an output distinct from the initially entered model.

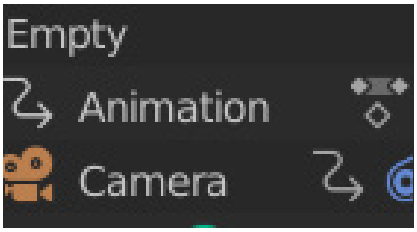
Upon tweaking the parameters and specifying the desired output path, a series of 100 images showcasing the rendered model was acquired. These images were then used to create a video following a fixed camera trajectory.











Bibliography

[1]Jong K D. Evolutionary computation[J]. Wiley Interdisciplinary Reviews: Computational Statistics, 2009, 1(1): 52-56.

[2]Mahoney M S. The history of computing in the history of technology[J]. Annals of the History of Computing, 1988, 10(2): 113-125.

[3]Burkholder L. The halting problem[J]. ACM SIGACT News, 1987, 18(3): 48-60.

[4]Gillespie T. The relevance of algorithms[J]. Media technologies: Essays on communication, materiality, and society, 2014, 167(2014): 167.

[5]Soare R I. Computability and recursion[J]. Bulletin of symbolic Logic, 1996, 2(3): 284-321.

[6]Wegner P, Goldin D. Computation beyond Turing machines[J]. Communications of the ACM, 2003, 46(4): 100-102.

[7]Hamkins J D, Lewis A. Infinite time Turing machines[J]. The Journal of Symbolic Logic, 2000, 65(2): 567-604.

[8]Gill III J T. Computational complexity of probabilistic Turing machines[C]//Proceedings of the sixth annual ACM symposium on Theory of computing. 1974: 91-95.

[9]Jones N D. Computability and complexity: from a programming perspective[M]. MIT press, 1997.

[10]Immerman N. Computability and complexity[J]. 2004.

[11]Brezeale D, Cook D J. Automatic video classification: A survey of the literature[J]. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 2008, 38(3): 416-430.