Index

1. Introduction

A genetic algorithm (GA) is a type of search and optimization based on biological processes. Solutions are encoded into a string of binary numbers, and these representations are treated as chromosomes. The GA replicates natural selection to evolve better solutions. The GA uses systematic processes, implemented in ways that yield random results, and have been applied in a wide range of domains. These unique outputs have also attracted attention in applications for creative purposes. GAs and evolutionary computation techniques have been applied to evolving digital arts, in addition to applications within traditional fields of engineering, medicine, science, and mathematics.

The challenge in using a GA to evolve art comes from the fitness function, or the criteria for evaluation. As human tastes vary from person to person, one approach is to use a human in the loop for evaluation to direct evolution, i.e., the human computes the fitness function. While this approach produces art that is a good match for the specific user, it does not reveal much for future implementations of a creative artwork-producing GA. In addition, the human in the loop significantly influences the speed of the evolution process. Another approach is to define the fitness function, based on the fundamentals of art and design. This approach aims to find characteristics of art that can be evolved to creative designs, without the need for continuous human interaction.

This paper uses an autonomous GA to create artistic designs, based on predetermined fitness functions. Aspects of art, such as color or symmetry, are focused on. For each experiment, one aspect is chosen as the criterion for evaluation. The GA evolves the images to better fit the chosen criterion. After the images have been improved, the designs are displayed graphically. Owing to the random element of the GA, the output of every experiment is unique.

After seeing this collection of geometric designs, we remembered something that We was going to do. Genetic art, or evolving pieces of art. The idea came to me when We was looking into what AI has been used to replicate. Something like creativity. We are quite into geometric art, so We wondered if We could replicate that using genetic algorithms.

2. Project Objective

Given a limited number of brushes placed on the canvas, how must we position, scale, rotate and color these brushes, to create a result which is visually as close to the original image as possible?

If you see this and think, we have no clue how to mathematically do this, then genetic algorithm, or any other search/ optimization is a satisfactory solution to the problem.

You can solve this problem in a more analytical way. If you are interested in alternatives that run faster and produce more consistent results, you can have a look at parts of some feature detection algorithm like SIFT.

- **AI as an Impersonator**

We began using AI to create art by first teaching it to understand and replicate our own art. The technique is called style transfer and it uses deep neural networks to replicate, recreate and blend styles of artwork. It identifies and combines stylistic elements of one image and applies them to another. No artistic or coding experience required.

Whether it's applied to paintings, photography, video, or music, the concept is the same: choose a piece of artwork whose style you want to recreate, then let the algorithm apply that style to a different image. Or, choose several styles of art and let the AI produce mash-ups that incrementally blend styles together

3. Methodology applied

a. Genetic Algorithms

Genetic algorithm is in its essence a search operation, modeled after evolution. In a search operation we look for the best answer to a problem. We obviously do not know the answer, but we have a way of measuring how fitting a given random answer is. The goal is to find the best possible answer we could. The best answer is one that has achieved the highest fitness score according to a function we define.

A good resource, which is way shorter is this one which goes in depth on all the terminology which We will be using. We will not be repeating whatever is there, in favor of writing things that are not there, so have a look at those.

In genetic algorithm terminology, an attempt at an answer was made up of a bunch of **genes**. The gene is the different parameters of your problem. When these genes are put together, they stand for an individual attempt at the problem, or a **population member.** A bunch of population members put together make up the **population pool.** The population pool is the collection of the random attempts the algorithm is making.

Here are the general steps for a genetic evolution algorithm:

- Population is initialized with **n** members. Each member gets a random set of genes
- Each member is assessed by a **fitness function**, which figures out how close the population member is to the desired answer
- Based on the fitness value of each member, a probability is assigned to a member, which figures out how likely it is to become a parent for the next generation
- In a **parent selection** phase, based on those probabilities, for each second-generation member, **n** number of parents are selected
- In a cross over, the genes of parents are combined to create a new solution as a combination of earlier ones.
- To not get stuck with only the combinations which were created in step one, from time-to-time genes are **mutated** and take random new values

b. Population Creation

First step. A population member in our case is an attempt at painting the picture. For convenience's sake, let us say we want to paint the image with only 120 brush strokes. Then a population member is a collection of 120 brush strokes layered on top of each other, which make a painting.

A brush stroke in our case is the gene. The gene is made up of the position, rotation, scale, color, and texture of the brush.

A group of population members, make up the population pool. We create them in a compute kernel on the GPU, using random numbers created from integer seeds.

To ease up the burden of the program, when the image is in black and white, we create only black and white brushes.

Our population creation is not over with just that. The genes (the brush strokes) cannot be assessed on their own for the fitness. That can only happen after they have been used to draw an image, which then gets compared to the original photo.

c. Fitness Function

I tried so many things and then stuck with the most basic approach. How do you compare two images with each other and decide how similar they are to each other? The easiest way, which We went for at the end is to compare the value of each pixel of the painted image with the original and take the distance between them as the error. The fitness is one minus this error.

Another approach which We tired, which did not turn out as expected is to compute the gradient image for both and compare the per pixel values on that. My thought here were We rarely want to know how well a pixel is doing. I would rather know how well pixels are doing in relation to each other. This made the fitness function too incoherent for the optimization and the results got worst.

The most important thing We learned here, especially for colored photos is, to not use RGB space. A lesson We relearn again, every time We work with procedural colors. RGB space is not perceptually uniform, which is an issue.

If you look at the color wheel made of the HSV space, you will realize that a unit of change in the green area makes barely any noticeable difference, while a unit of change in the red area shifts through major hues such as red, orange, and yellow.

This makes the fitness function, which calculates the distance between the RGB values per pixel, super precise in the green areas and unstable in the red.

Once you have a fitness function per pixel, you need to sum it up to get one value for the entire image. We do all these steps in the fitness function compute over several kernels. This is not the best way to sum up a buffer in GPU, since it is not using the parallel power of the GPU. The typical way to do this is a parallel reduction, which We did not implement since it was more work.

d. Fitness to Probability

This stage is identical to any other genetic evolution algorithm. The goal is to convert the fitness values to something we can use to randomly pick parents for the next generation.

The easiest way would be to use the fitness directly as probability. Mapped one to one. Next step would be to convert these probabilities to cumulative values, to be able to use it for efficient sampling. We summed up the probability in an added buffer holding only cumulative probabilities in compute shader.

If you do just the above, you will have the issue that your images will take forever to converge to anything. We mean forever. The fitness function is not aggressive enough to differentiate strongly between bad painting tries and good ones.

To fix this issue, the first thing We tried was a power function. This is a typical strategy to artificially increase contrast between probabilities. This helps to speed up getting to the final image, but not by much.

To speed up the process further, we decided to make my fitness function context aware. The fitness value one is unattainable, it is the perfect image. The fitness value zero is so bad, that any random attempt would score higher. So why should we take these two irrelevant extremes to assess the probability of a member becoming a parent?

First, we find out what the worst and best members in this generation are. Then We re normalize the fitness of all members and remap them to the range of worst to best member of the population, so that the worst gets fitness zero and the best fitness one. This way the worst member has zero chance of being picked as a parent and the best has an extremely high one. This is like the Level image processing panel in Photoshop. Which remaps the pixel values to new range, to artificially maximize contrast.

This change to the fitness function drastically increases the speed of the search. So, what is the catch? There is always a catch. More aggressive fitness functions mean faster convergence to the best solution around the area you are searching. But it also means the highest probability of getting stuck in local minima, which you cannot get out of.

I think this a more general gradient descent problem. We did not investigate this much, as We found a solution quickly, but one of the things We thought about trying was to increase mutation rate and make the fitness function more liberal as a function of the change of fitness. So, if in a local minimum, start trying out the more liberal options and try to get out of the pit.

e. <u>Parent Selection Cross Over and Mutation</u>

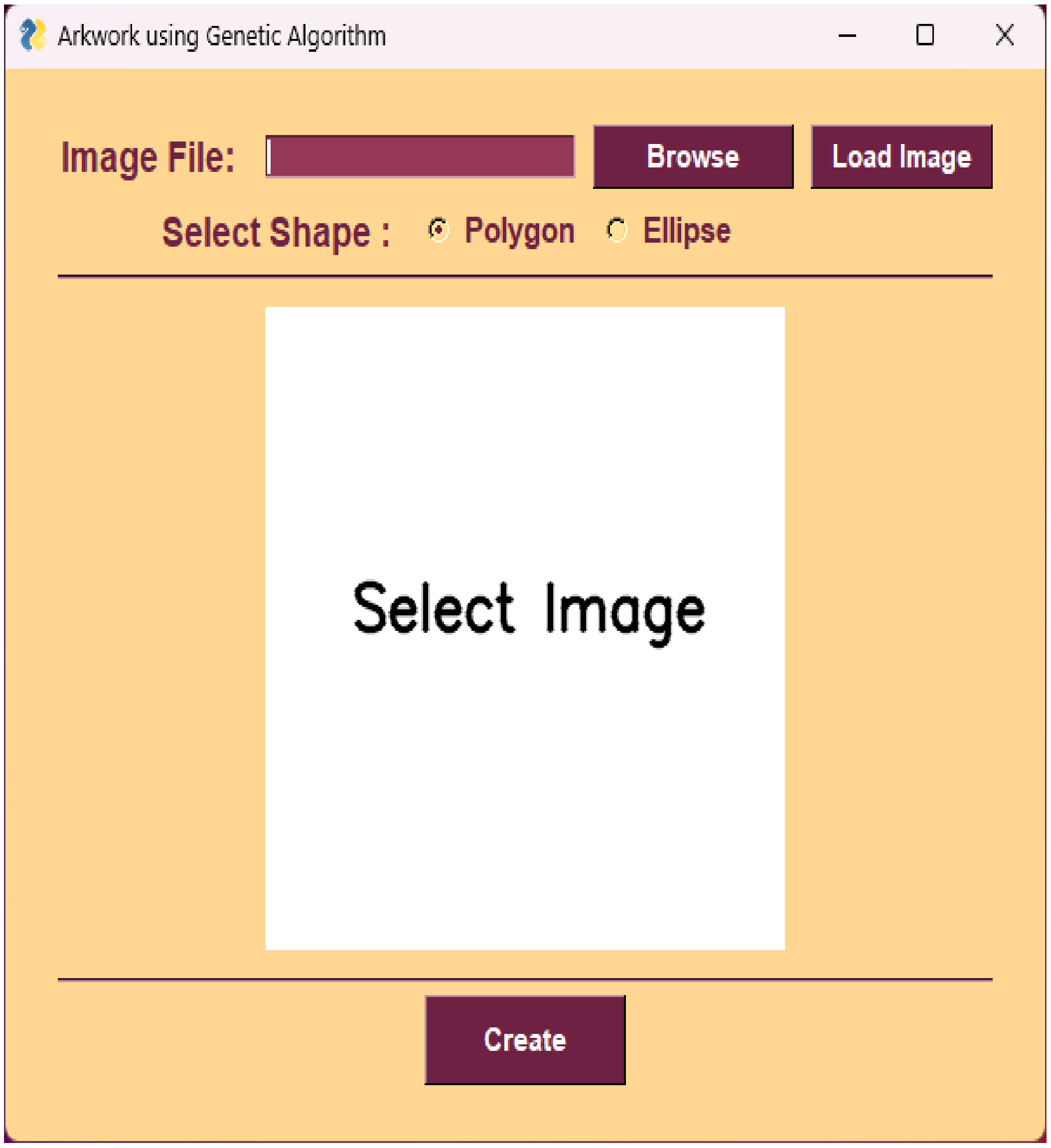
I am grouping these together, though they are each their own compute kernel, because there is not much to be said about them. Based on the cumulative probabilities We calculated in the earlier stage, we select two parents per member for the second generation, and We take half the brush strokes from one parent and combine it with the other half from the other parent to create the second generation. Then based on the mutation probability specified in the balancing parameters, we sometimes randomly change a brush stroke in the child.

Higher mutation rate means faster convergence. Because it will try out different solutions faster. However, if the mutation rate is too high, your population will become unstable. There will not be enough generations for the changes to settle in, and the best genes to become dominate in the population.

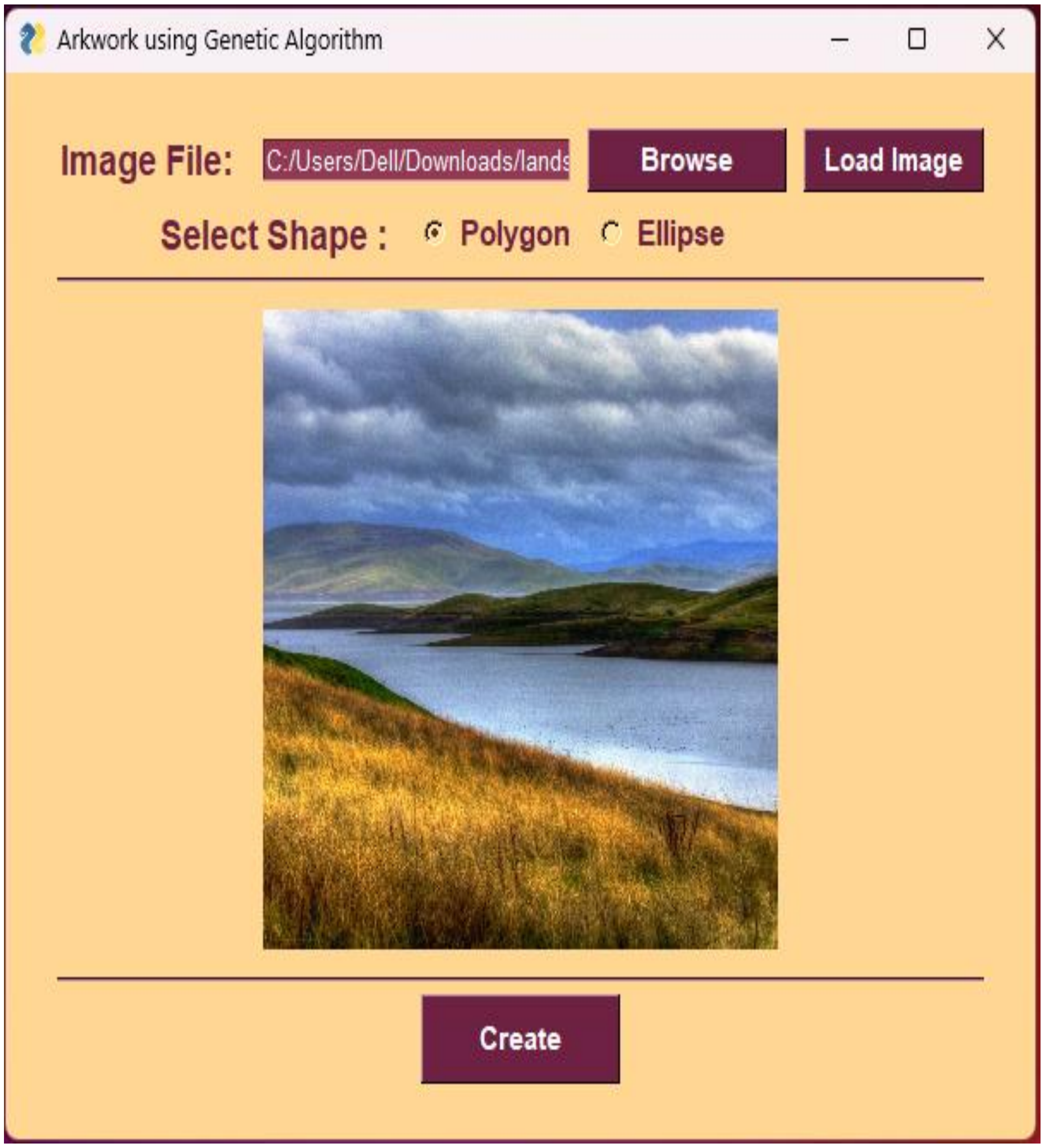
Very straight forward. This is a loop that repeats itself. The second generation becomes the parent of the third, third of the fourth and so forth.

4. Input & Output

- Loading Page



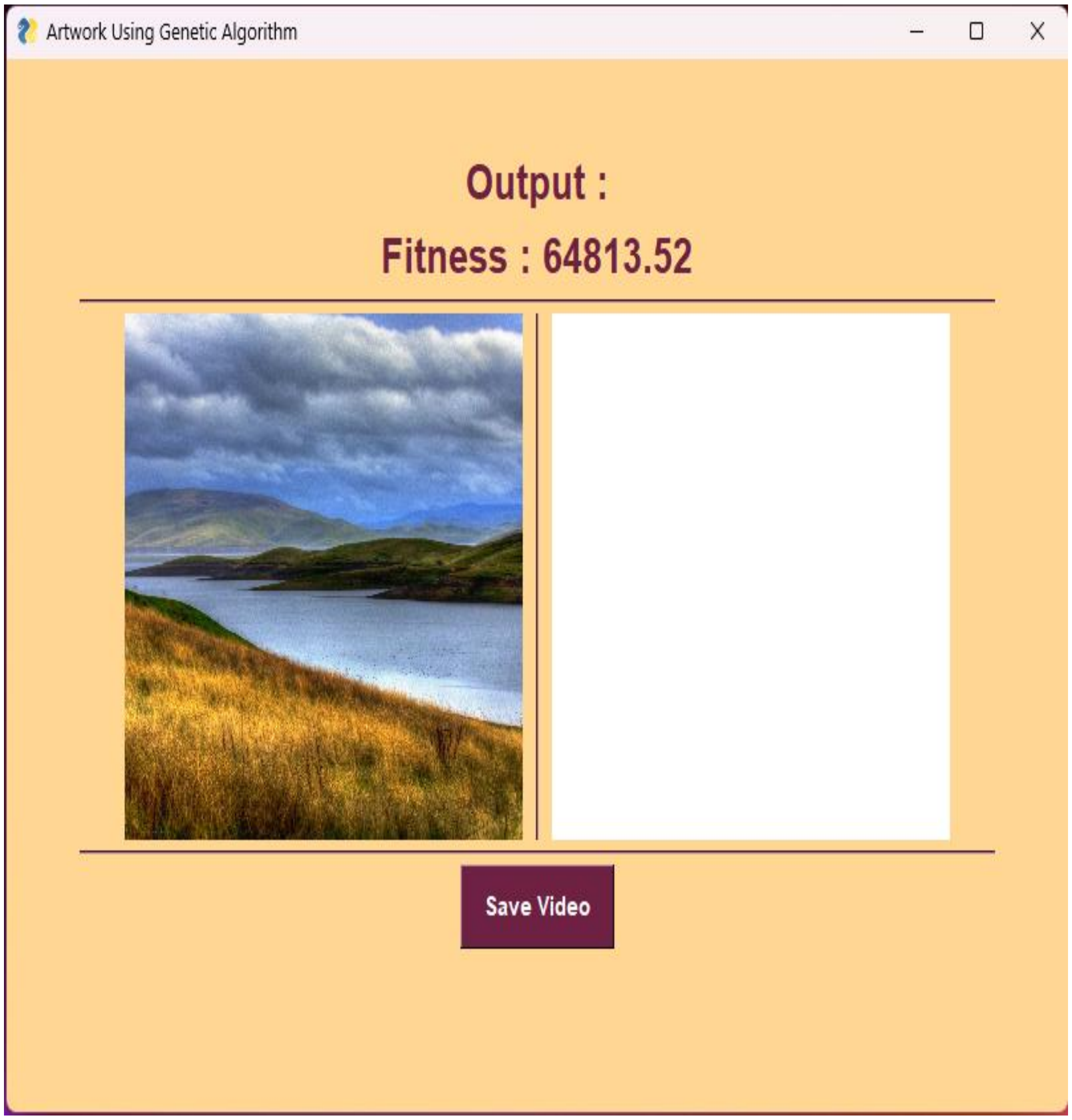
- Image Insertion Page



C:/Users/Dell/Downloads/lands

Select Shape : [✓ Polygon] Ellipse

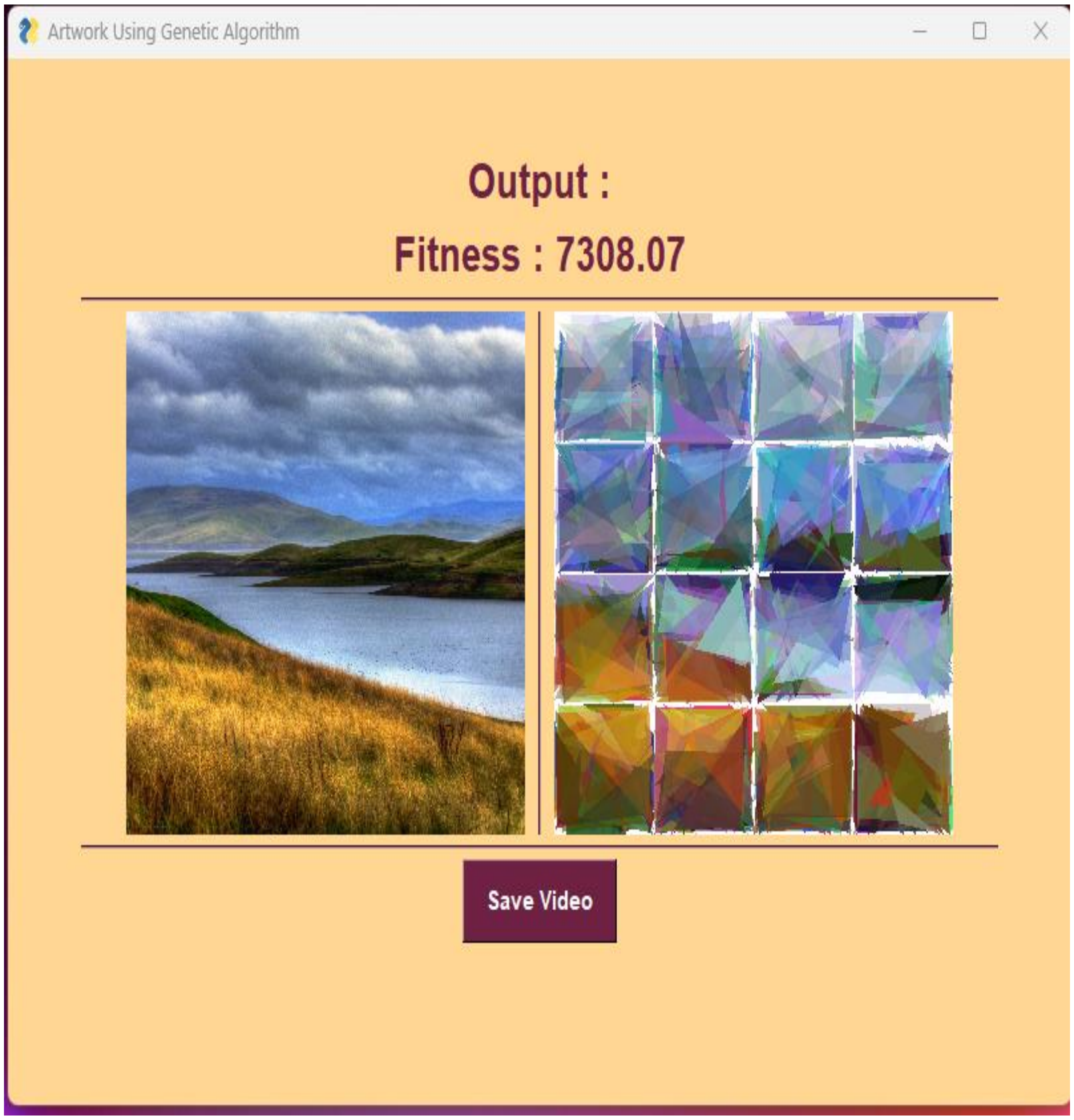
- At 0 Generation mean Starting



Output :

Fitness : 64813.52

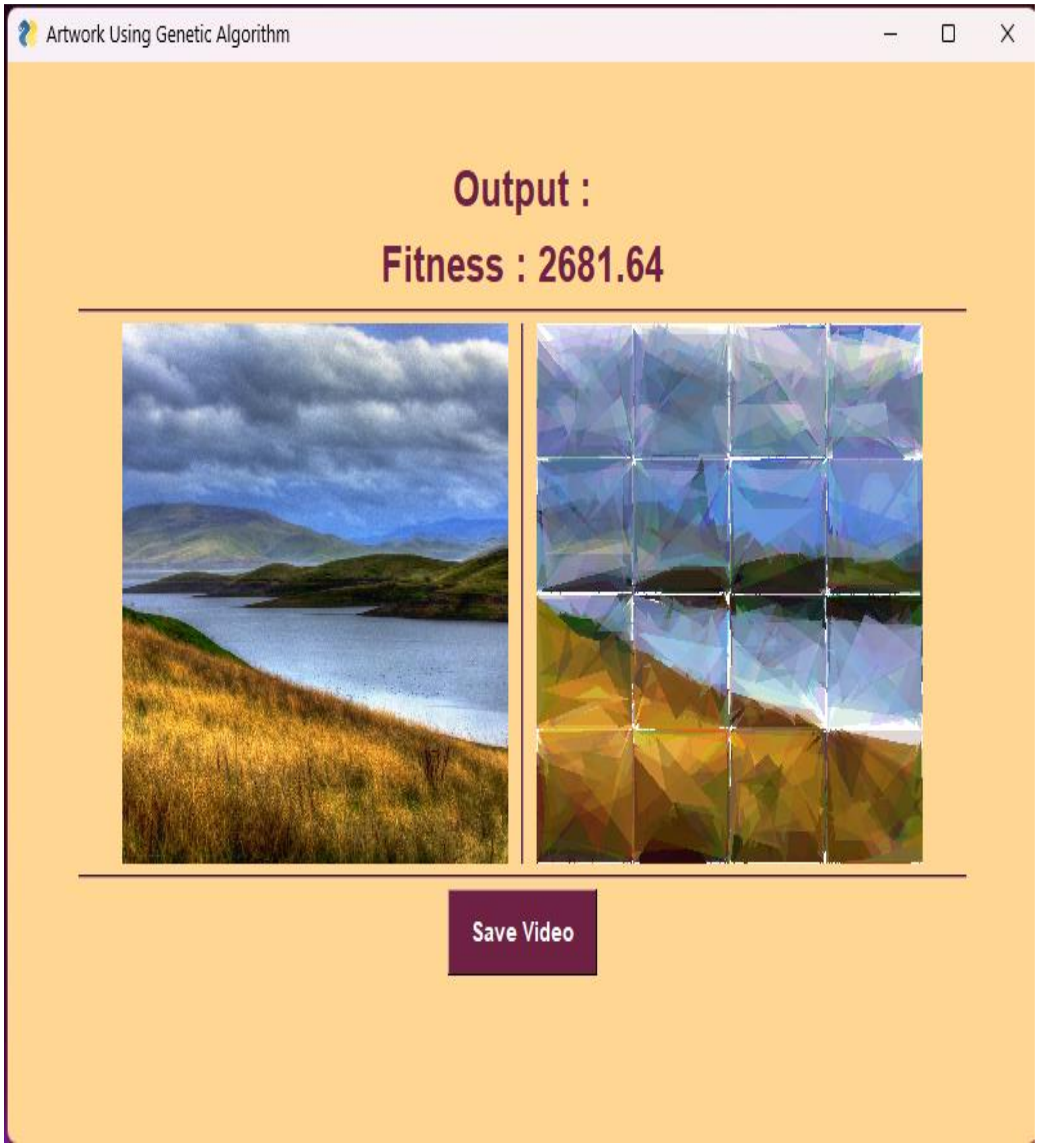
- At 1000 Generation Completed



Output :

Fitness : 7308.07

- At 3000 Generation Completed

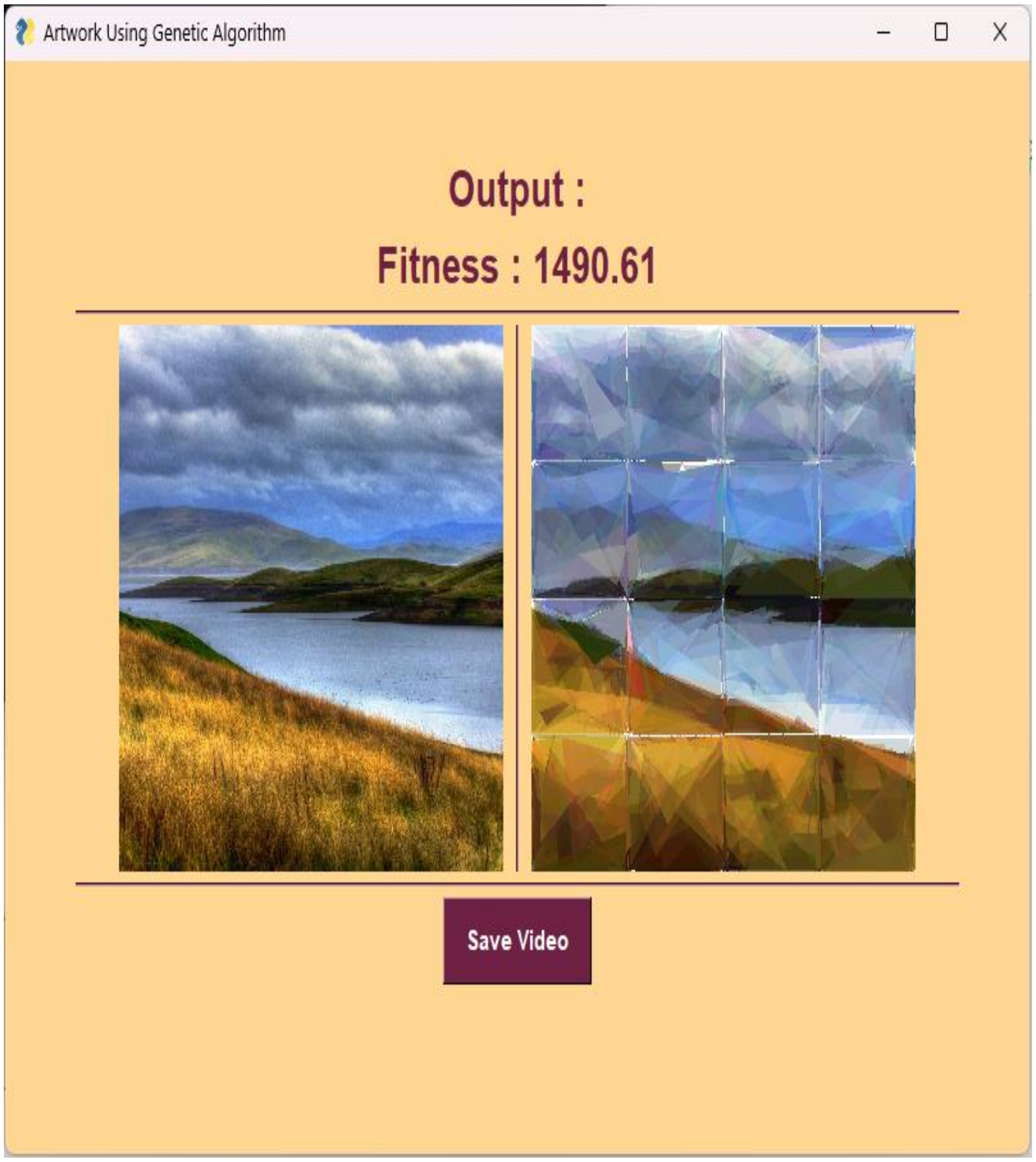


Output :

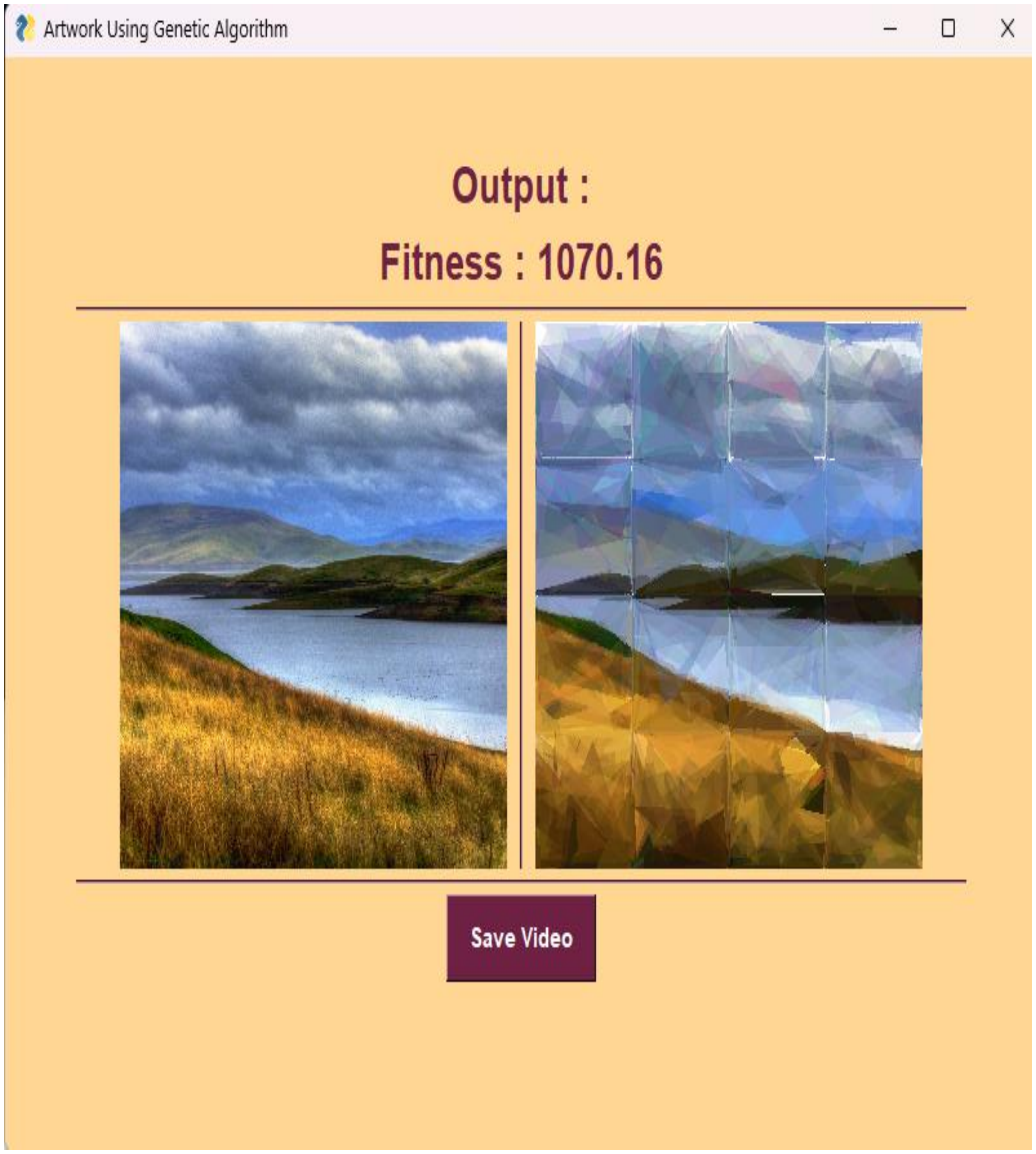
Fitness : 2681.64

S

- At 5000 Generation Completed



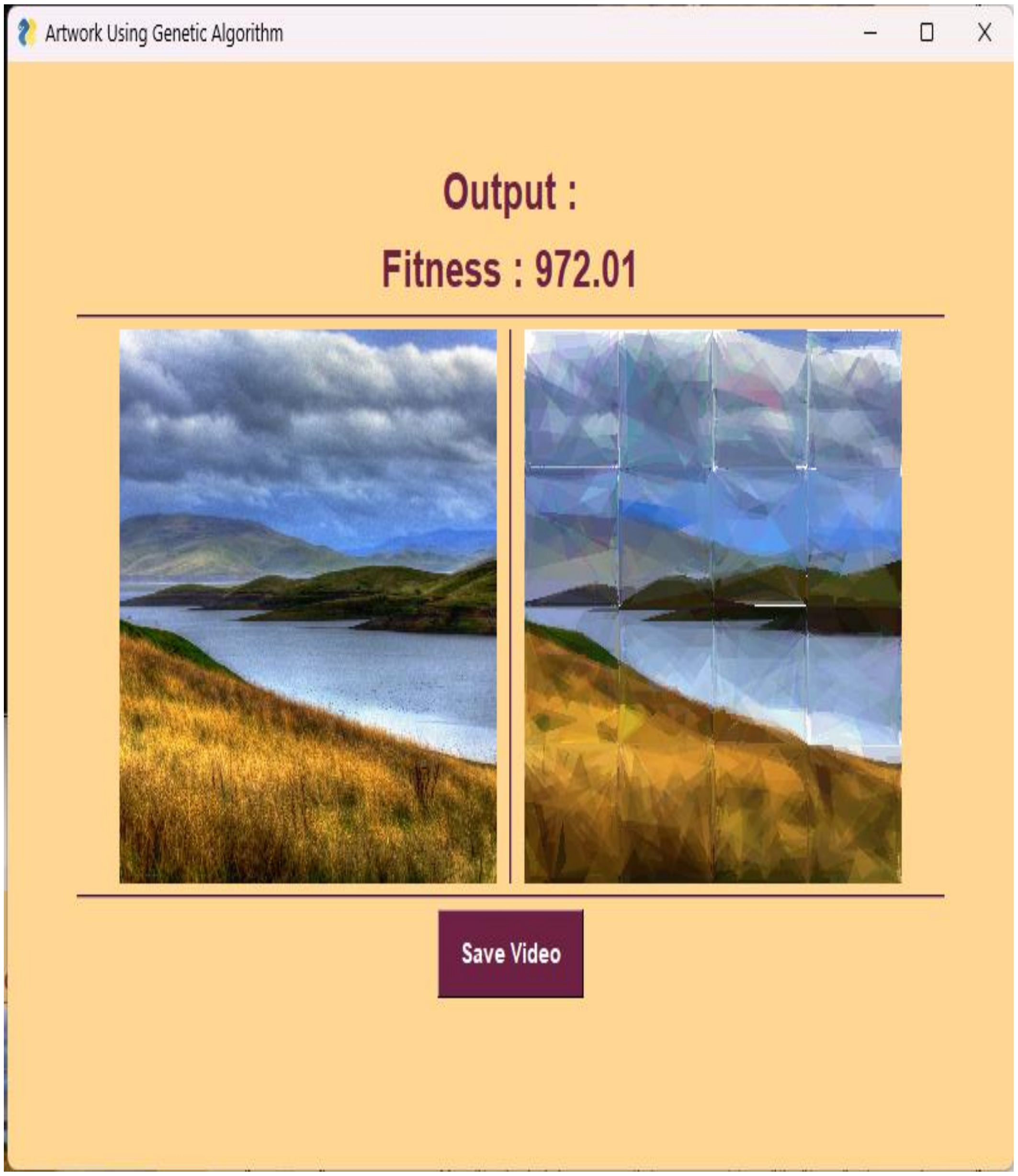
- At 10000 Generation Completed



Output :

Fitness : 1070.16

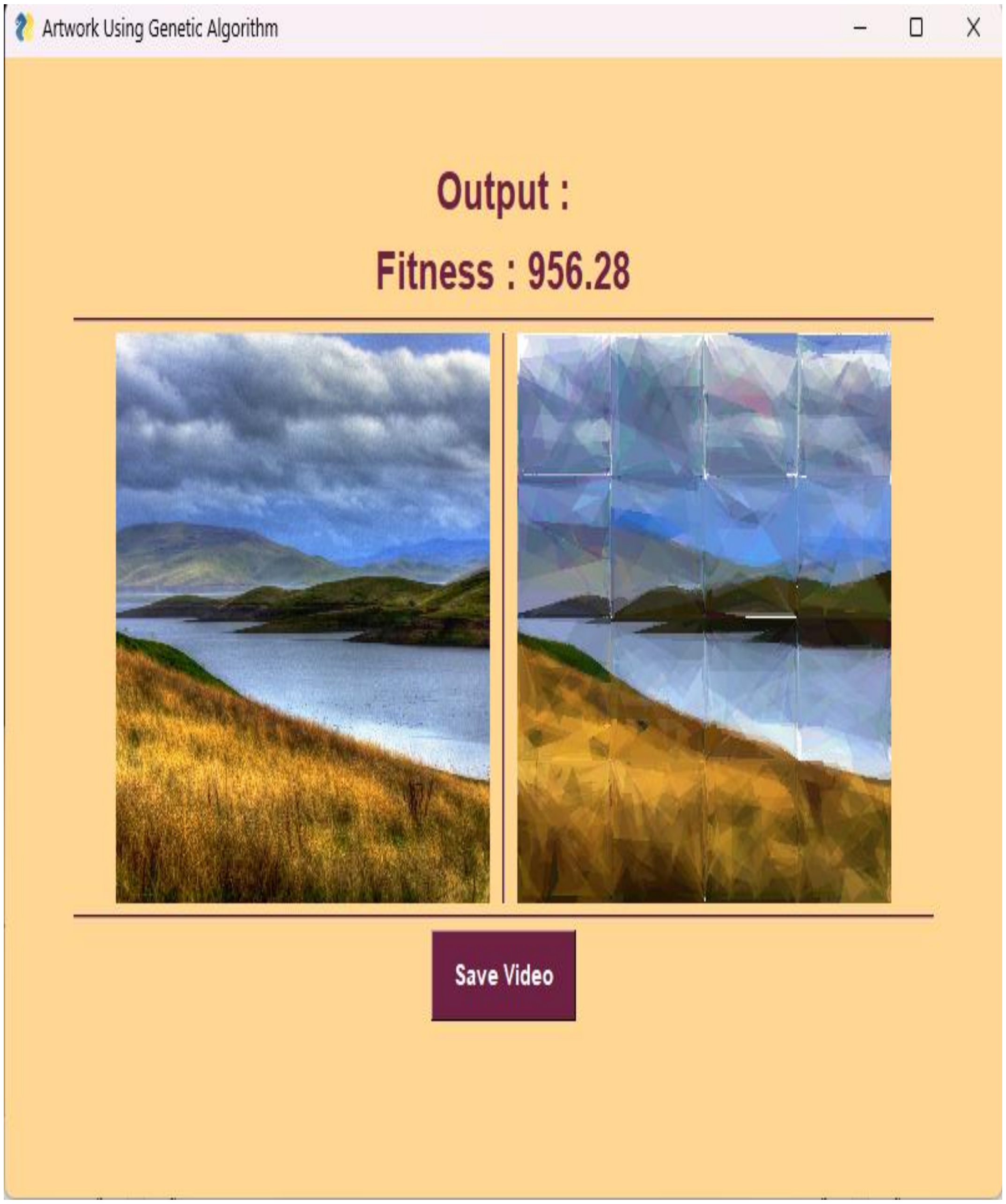
- At 20000 Generation Completed



Output :

Fitness : 972.01

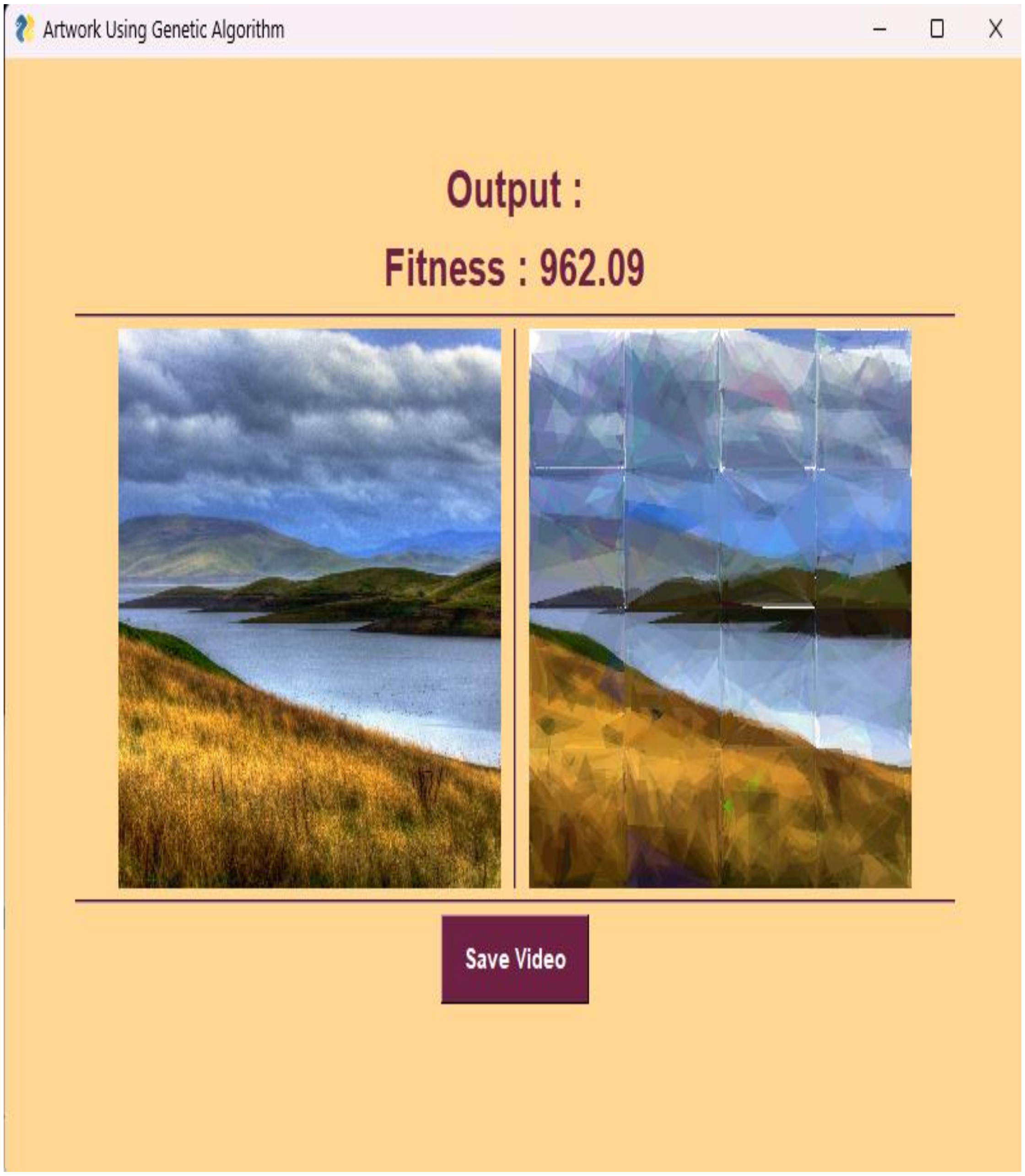
- At 50000 Generation Completed



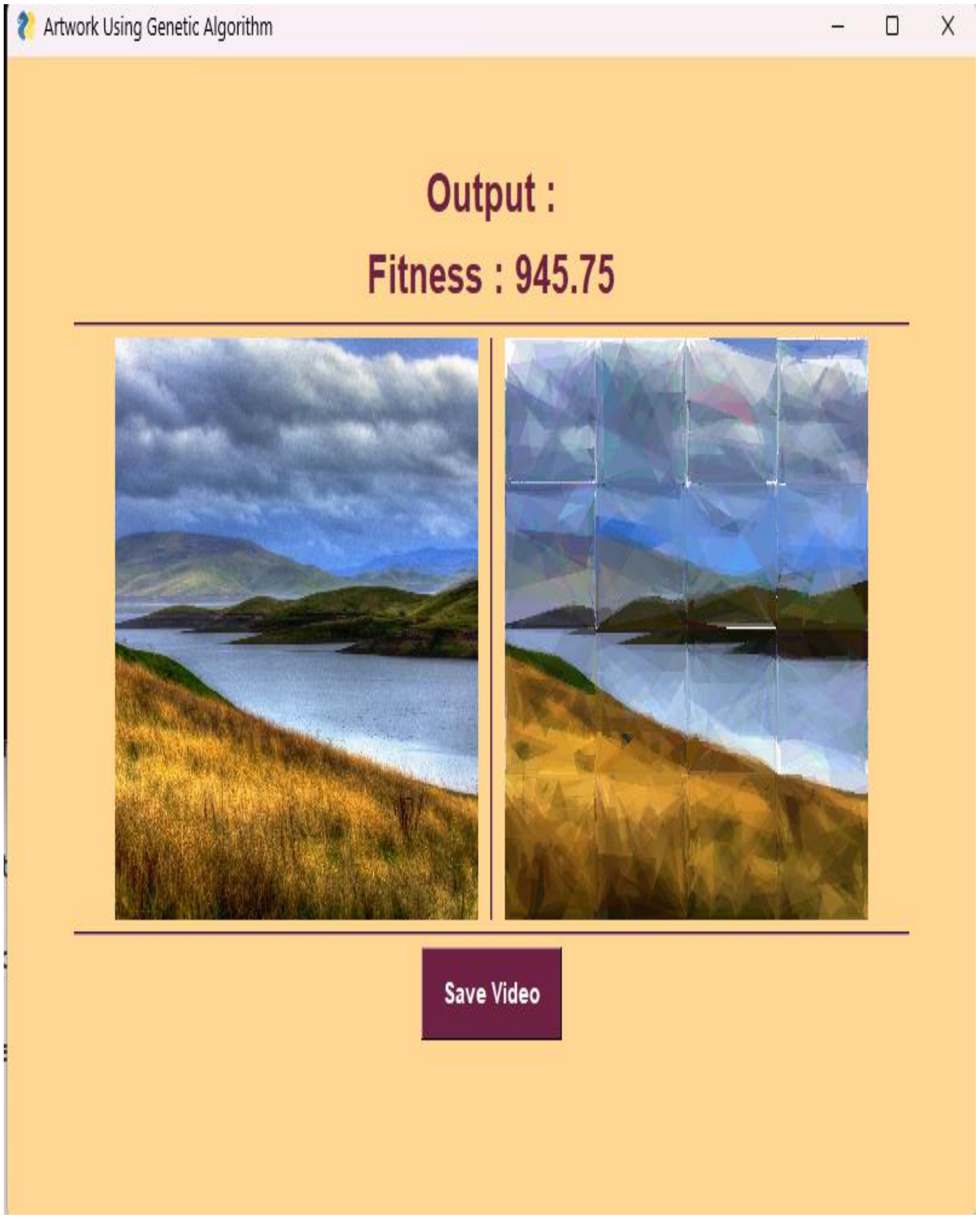
Output :

Fitness : 956.28

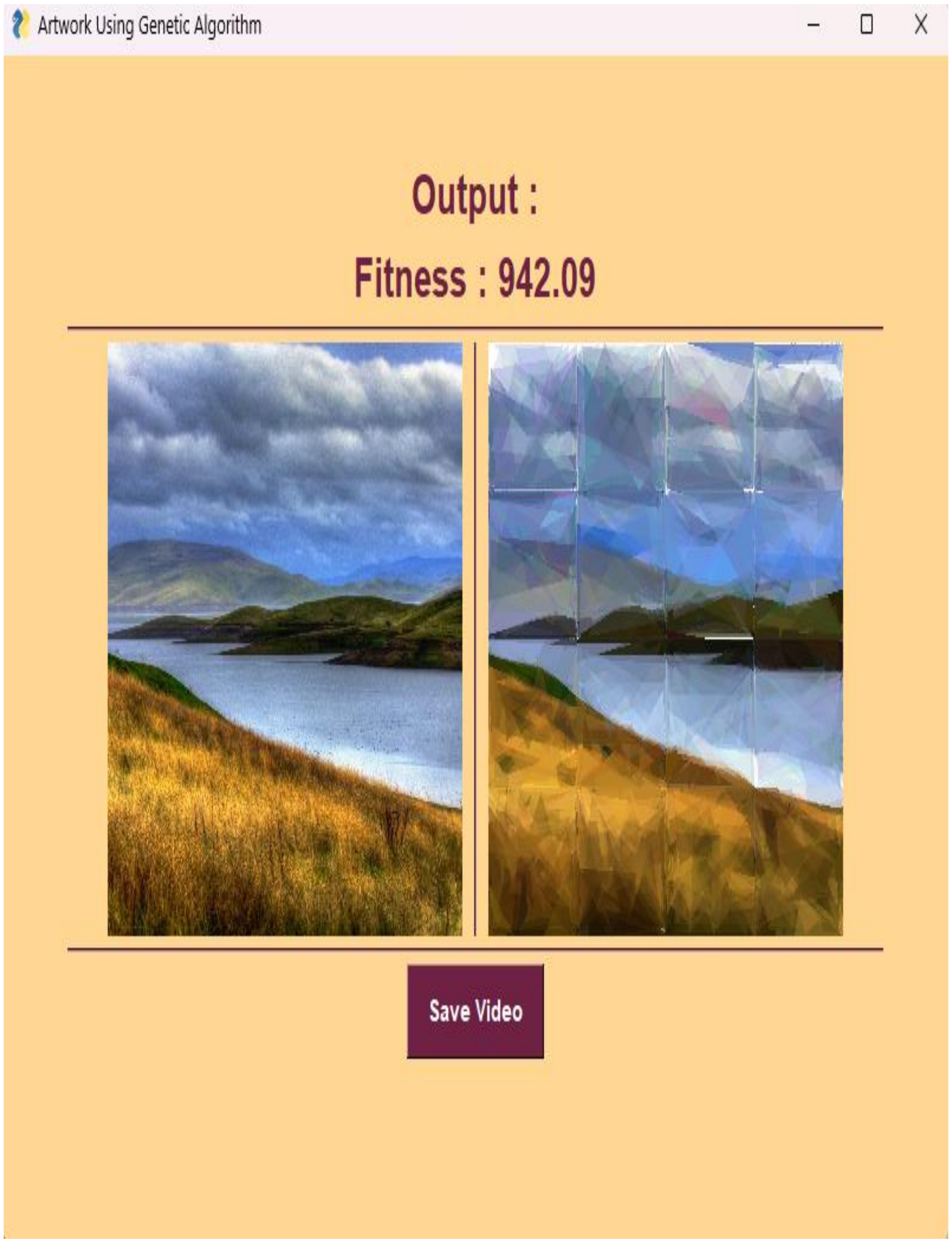
- At 75000 Generation Completed



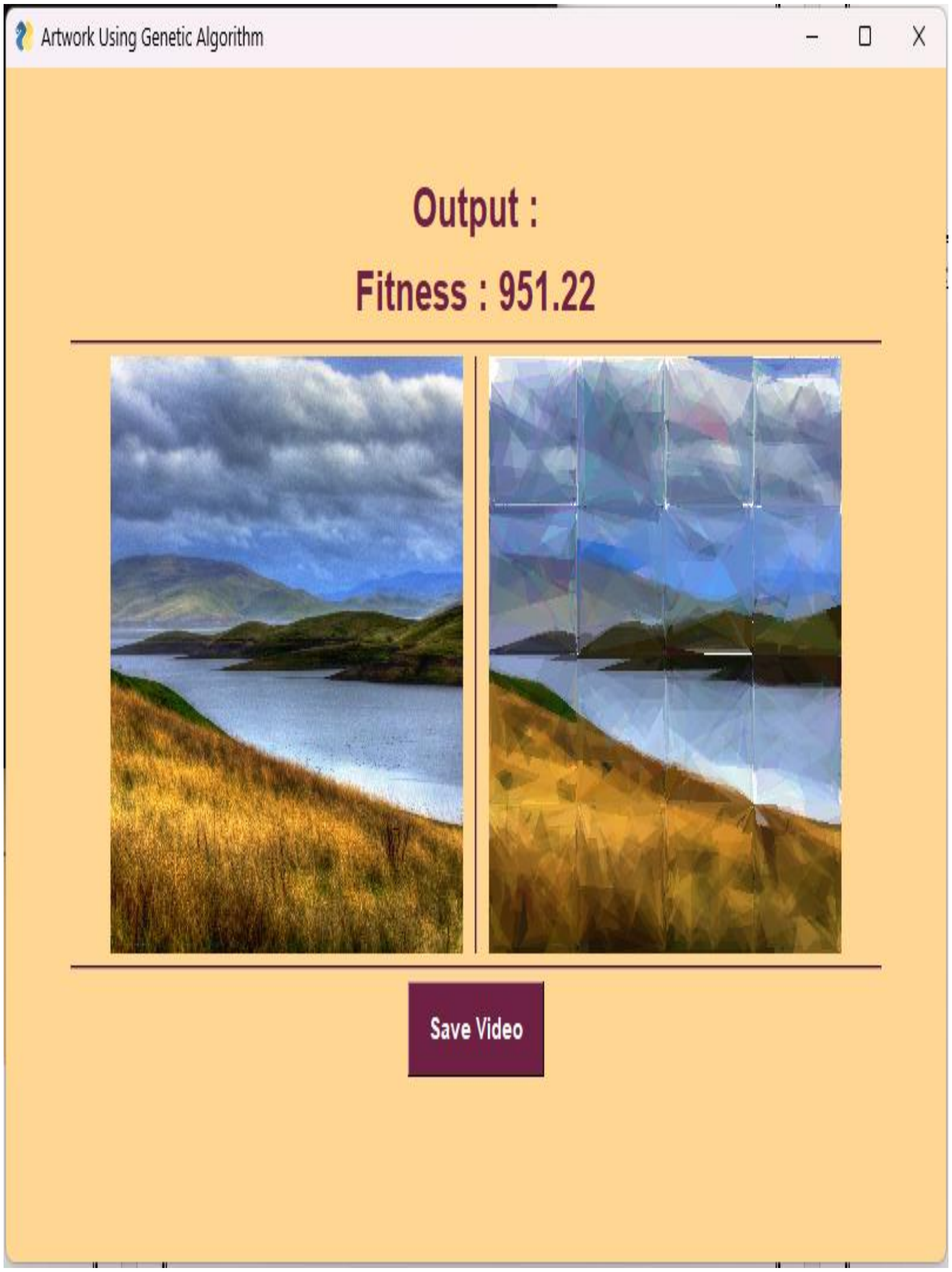
- At 100000 Generation Completed



- At 100000 Generation Completed



- Final Output



Around 175756 Generation with around 88% Similarly

5. Bibliography

a. Conclusion

There is still work left that can be done. Too much actually. I might come back to this one day and try out those ideas. For example, by adjusting the fitness function slightly, you can very differently result. Here is a fitness function that only cares about accuracy in value.

The purpose of this paper is to autonomously evolve images, by using predefined fitness functions rather than having a human in the loop. This paper takes steps toward the task of finding objective measures in art and automating the evolution of designs according to different sets of criteria chosen by different people. More fit individuals have images that are representative of the fitness functions used to evolve them. The images are indeed recognizable for the intended evolution. Furthermore, the high fitness results prove the success of using a hybrid approach between algorithmic and human in the loop fitness functions. By combining the strengths of the two – incorporating human choice at the beginning, then running efficient algorithmic fitness functions – the results show that it is possible to automate the evolution of digital art. The underlying approach can be extended to other applications to autonomously evolve creative works

The memory issue will be addressed to manage more complex designs. Possibilities on the hardware side include obtaining a more powerful machine to use to run the GA or looking into parallelism.

The work done in this paper can lead to developing a more robust system that can, without the need for human intervention, evolve digital art with human-like creativity. To do so, it will be necessary to understand how people evaluate art. The tastes of individual people will differ in the specifics, but it may be possible to find the parameters of preferences. For example, one person may like symmetry, and another may dislike it; though they disagree on which is better, symmetry is a criterion used to judge art. This paper focused primarily on color and symmetry. After studying art and finding a more complete list of characteristics, then more complex and engaging art can be generated.

b. Reference

1) Daniel Schiffman’s video series on genetic algorithms: https://www.youtube.com/playlist?list=PLRqwX-V7Uu6bJM3VgzjNV5YxVxUwzALHV
2) Same as above but as text for whoever prefers reading: https://natureofcode.com/book/chapter-9-the-evolution-of-code/
3) Post on pseudo random number generation by Nathan Reed: http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/
4) Article by Nvidia on parallel reduction in GPU: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
5) Sampling elements from a list given their probability: https://stackoverflow.com/questions/38086513/selecting-random-item-from-list-given-probability-of-each-item
6) Post by Sebastian Altona on maximizing Optimizing GPU occupancy and resource for large thread groups: https://gpuopen.com/learn/optimizing-gpu-occupancy-resource-usage-large-thread-groups/
7) Microsoft documentation on compute shader group dispatch numbers and thread numbers per group: https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-dispatch
8) Kostas Anagnostou explaining best practices on the number of threads to dispatch per group: https://twitter.com/KostasAAA/status/1274359186185428994?s=20
9) Nice slides on down sampling and gaussian blur: http://www.cs.toronto.edu/~fidler/slides/2015/CSC420/lecture5.pdf
10) A gaussian blur implementation in Unity done in shader and graphic pipeline: https://danielilett.com/2019-05-08-tut1-3-smo-blur/
11) Entry on the use of append buffers in Unity: https://docs.microsoft.com/en-us/windows/win32/direct3d12/indirect-drawing
12) Microsoft documentation on the structured of indirect drawing argument buffer: https://docs.microsoft.com/en-us/windows/win32/direct3d12/indirect-drawing