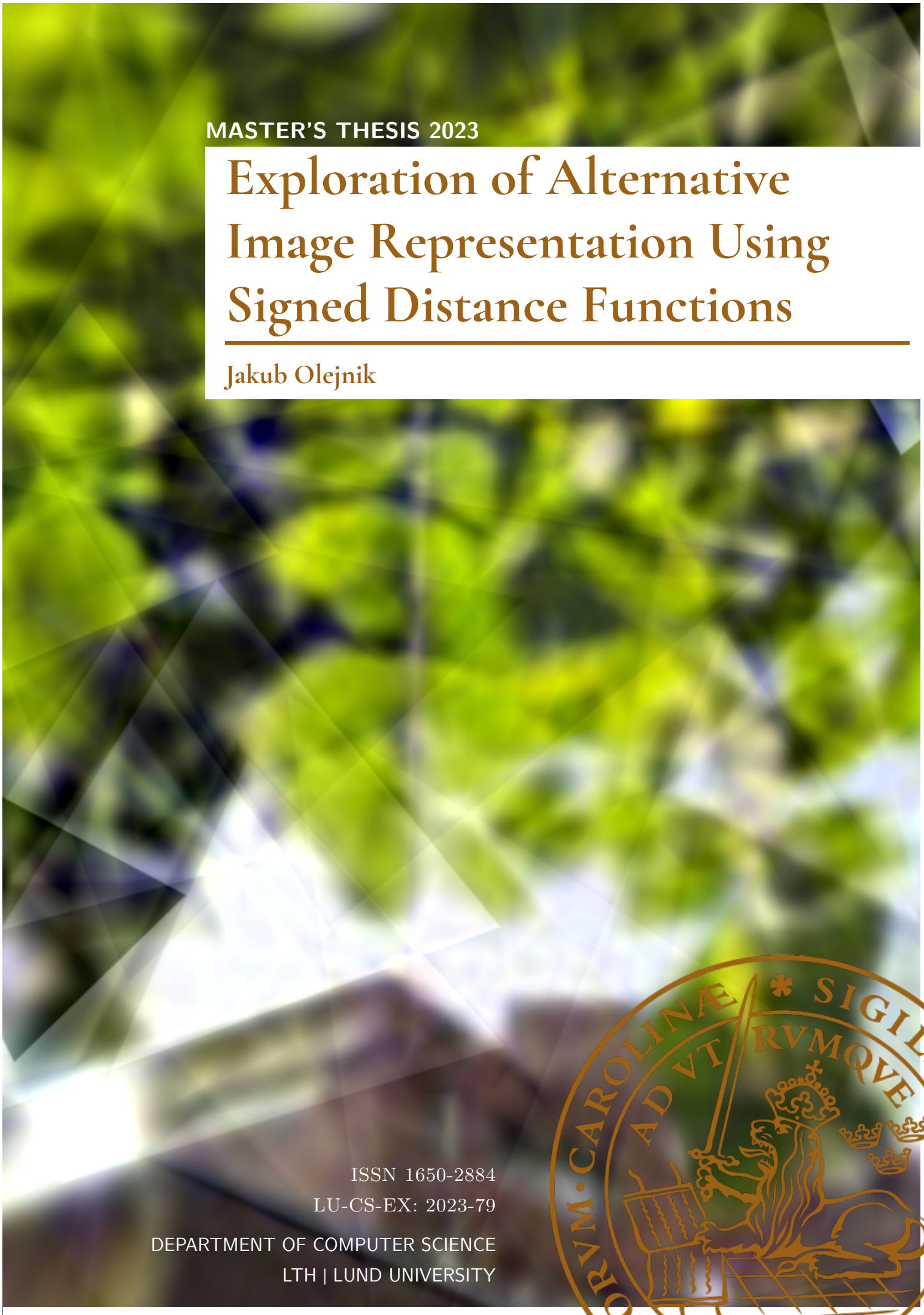


MASTER'S THESIS 2023

Exploration of Alternative Image Representation Using Signed Distance Functions

Jakub Olejnik

ISSN 1650-2884
LU-CS-EX: 2023-79

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-79

Exploration of Alternative Image Representation Using Signed Distance Functions

Jakub Olejnik

Exploration of Alternative Image Representation Using Signed Distance Functions

Jakub Olejnik
ja3664ol-s@student.lu.se

June 2, 2023

Master's thesis work carried out at
SEED, Search for Extraordinary Experiences Division, Electronic Arts.

Supervisors: Michael Dogget, michael.doggett@cs.lth.se
Chris Lewin, chlewin@ea.com
Jon Greenberg, jongreenberg@ea.com

Examiner: Mattias Wallergård, mattias.wallergard@design.lth.se

Abstract

This project leverages the core concepts of differentiable rendering to find alternative representation of images using geometric primitives rendered with Signed Distance Functions. By utilising gradient-descent-based methods it is possible to find optimal parameters of rendered primitives to create high-quality reconstruction. Moreover, this research serves as a foundation for extending the principles and methodologies explored to 2D space. While the project primarily focuses on 2D image representation, the insights gained can be applied to similar tasks in the field of 3D rendering.

Keywords: computer graphics, SDF-based rendering, machine learning

Acknowledgements

I would like to thank my university supervisor Michael Dogget for introducing and guiding me through the world of computer graphics, and all of his support during this project.

I would also like to thank SEED's Konrad Tollmar and Future Graphics team, especially Chris Lewin, Jon Greenberg and Colin Barré-Brisebois, for all their guidance and captivating conversations that made this project possible.

Contents

Chapter 1
Introduction

Exploration of different representations of data structures allow programmers to find solutions that introduce a new level of abstraction, optimise time and space complexity, conserve required memory, or tailor structures to specific needs. This is especially vital in the field of computer graphics, where rendering performance needs to be high, while maintaining low memory consumption, and allow for efficient processing of complex graphical data. With this idea in mind, we undertake the challenge of finding an alternative representation of digital images, which play a central role in various applications such as video games, image processing, and computer-aided design. In this chapter, we provide an overview of the project and outline its scope and contributions.

1.1 Project Scope

The thesis investigates how the principles of differentiable rendering can be applied in the process of image reconstruction using geometric primitives. To achieve this, we explore various aspects, including error metrics for measuring image reconstruction accuracy, optimisation algorithms for finding optimal parameters, and the intricate characteristics of Signed Distance Functions. By building a thorough understanding of all these components, the project aims to enhance the quality and efficiency of the 2D image rendering by addressing the following research questions:

- How many geometric primitives are needed to reconstruct a digital image?
- Can gradient descent be effectively utilised for finding optimal parameters of primitives for such reconstruction?

By answering these questions, we want to provide insights into the feasibility and effectiveness of differentiable rendering techniques for image representation using geometric primitives.

1.2 Contributions

Through this thesis, we hope to contribute to the advancement of alternative image representation techniques. By utilising machine learning, different error metrics, and various SDF characteristics, we seek to find novel ways for creating and storing computer generated imagery, which allow for versatile rendering and visualisation possibilities.

Chapter 2
Background

2.1 Computer Graphics

Computer graphics refer to various visual content, which is created using computer software and hardware. They are used in a wide range of fields, including animations, visual effects, video games, architecture, and data visualisation. In this work, we focus on 2D computer graphics, more specifically on digital images and their representation.

2.1.1 Digital Image

Digital images can be divided into two types:

Raster images, also known as bitmap images, consist of pixels arranged in a 2D grid, where each pixel stores colour information. Depending on the pixel format, colour data can have a variable number of bits, which is known as colour depth. Raster images are resolution dependent, meaning any change in scale will result in a loss of quality.

Vector images are products of mathematical equations that describe lines, shapes and colours. With the change of resolution, these equations can be recalculated to maintain sharp quality, unlike raster images.

2.1.2 Colour Representation

The most widely used colour representation in computer graphics is a combination of red, green and blue values (RGB). Each colour channel is represented by 8 bits, which store values ranging from 0 to 255. This representation allows for the creation 16,581,375 unique colours. In some cases, a fourth channel is introduced for transparency value called alpha (then RGBA).

Other colour spaces, such as hue, saturation, and value (HSV), and cyan, magenta, yellow, and key (CMYK) can also be used. Nevertheless, RGB is the most common representation and is the one we are using in this project.

2.1.3 Colour Blending

Colour blending refers to techniques used to combine the colours of multiple layers or images. The choice of blend mode determines the process of composition of the colours, whether through transparency, addition, multiplication, or other operations. Different blend modes result in varying visual effects. Presented below are concise explanations of blend methods that are relevant to our work.

Alpha Blending

Alpha blending, as the name suggests, uses alpha value of each pixel to determine how two layers will be combined. The alpha blending equation (Eq. 2.1), also known as the *OVER operator*, calculates the weighted average of the RGB colours of the background C_B and the foreground C_F layers. This implies, that when considering multiple layers, we have to evaluate them in a specific order, as this is an order-dependent operation.

$$C = \alpha_F \cdot C_F + (1 - \alpha_F) \cdot C_B \tag{2.1}$$

This blending technique enables the creation of smooth transitions and gradual fading between layers, making it suitable for creating transparent or semi-transparent effects (Fig. 2.1a).

Additive Blending

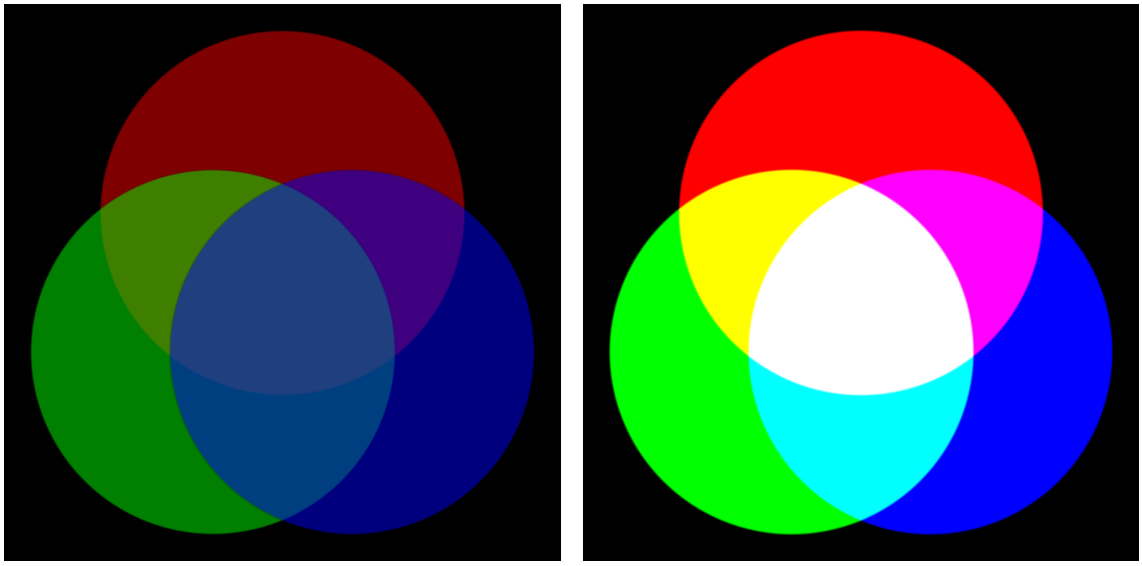
In additive blending the colour values of the layers are directly added together (Eq. 2.2). Each layer's colour contributes fully to the final colour, resulting in a brighter and more intense image (Fig. 2.1b). Unlike alpha blending, this technique does not take into consideration transparency information. Moreover, given that addition is commutative, this method is order-independent.

$$C = C_F + C_B \tag{2.2}$$

Additive blending is based on how humans perceive light and colours. It is commonly used for combining elements with additive properties, such as light sources or glowing particle effects.

2.1.4 Rasterisation

Rasterisation of 2D primitives is the process of converting the basic geometric shape, such as lines, curves or polygons, into a raster image. This requires determining which pixels are covered by the primitive and then assigning them colour values (Fig. 2.2). To detect which pixels belong to the primitive, we can loop over the pixels in the image and perform an inside-outside test. This process can be optimised by introducing *Axis Aligned Bounding Boxes*, in order to minimise the amount of pixels we need to perform tests on.



(a) (b)

Figure 2.1: Alpha (a) and additive (b) blending side by side

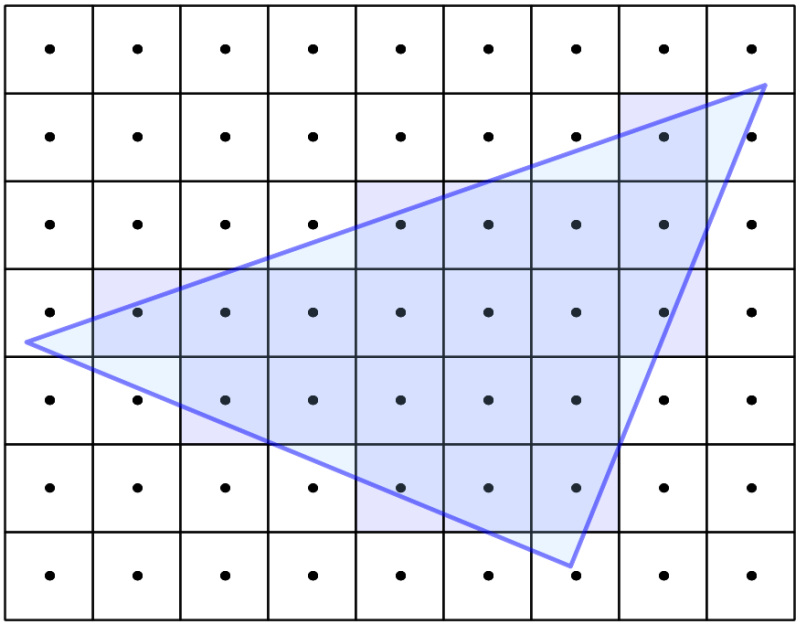


Figure 2.2: Pixels covered by a triangle

2.1.5 Signed Distance Functions

Signed Distance Functions, or SDFs as we will be referring to them from now on, are mathematical functions that measure the distance between a point in space and a geometric object. The *signed* component refers to the fact that the distances have different signs if they are inside (negative), outside (positive) or on the surface (zero) of the object (Fig. 2.3). The final result

is a field with a gradient of unit length everywhere, which means that SDFs are fully differentiable. The direction of the gradient for a surface is exceptionally useful and important in the context of rendering complex scenes, where it can be used for lighting calculations and aligning objects to surfaces.

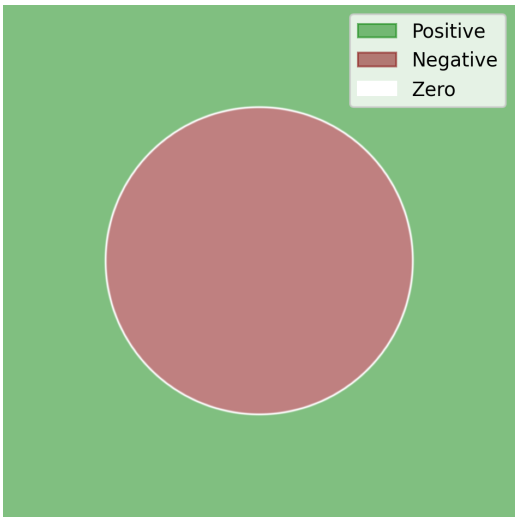


Figure 2.3: A visualisation of evaluated SDF for a circle

SDFs can be used to provide an implicit representation of any geometric object, that can maintain a sharp quality with any change in scale due to the fact they are described by mathematical equations, which can be recalculated. Moreover, they can be combined using Boolean operations, such as union (Fig. 2.4a), subtraction (Fig. 2.4b), and intersection. (Fig. 2.4c) It is possible to create complex shapes by fusing a set of base primitives, just like *Constructive Solid Geometry*[1].

(a) **(b)** **(c)**

Figure 2.4: Examples of union (a), subtraction (b), and intersection (c) between a circle and a box

They are also not limited to 2D space, they can be easily extended to 3D by performing extrusion or revolution on the 2D shape representation. SDF's versatile nature and various benefits are what make them so intricate, and useful in many areas of computer graphics: font rendering [2, 3], procedural modelling [4], or global illumination [5], just to mention a few.

2.2 Optimisation

An optimisation problem is a mathematical process in which the goal is to find the best solution among all possible solutions, given a set of criteria or constraints. In other words, we want to minimise or maximise some objective function, which is often referred to as *loss* in case of minimising, or *fitness* in case of maximising. There are several algorithms that can be used to solve optimisation problems, and below are brief descriptions of methods relevant to this work.

Gradient Descent

An iterative optimisation algorithm used to minimise the value of a function by iteratively adjusting the parameters of the function. Gradient Descent works by calculating the derivative of the function with respect to the parameters at a given point, also known as gradient. The gradient points in the direction of the steepest increase of the function, so the negative of the gradient points in the direction of the steepest decrease. In each iteration of the algorithm, the parameters are updated by subtracting the product of the learning rate and the gradient from the current value of the parameters. The learning rate controls the step size taken in the direction of the negative gradient, and is typically a small positive value in the range 0 to 1.

If the problem can not be represented with differentiable equation, the gradients can be obtained using the *finite difference* method. A derivative is then approximated by the expression in the form of the difference between function at point x and $x + h$, and divided by h (Eq. 2.3). Where h has a small fixed non-zero value.

$$\frac{d}{dx}f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+h) - f(x)}{h} = \frac{\Delta_h[f](x)}{h} \tag{2.3}$$

Adam

Base Gradient Descent can have trouble finding a global minimum if the learning rate is not set properly, too small step size means slow convergence, and too big might lead to overshooting. Adam is a more advanced gradient descent variant that overcomes this issue by incorporating both momentum and adaptive learning rates, making it more efficient. It adapts learning rates for each parameter based on their historical gradient statistics, which results in faster convergence and better generalisation [6, 7].

2.2.1 Machine Learning Libraries

Machine Learning libraries are software tools that provide pre-built functionality that streamline the process of development of various machine learning models, including those to solve optimisation problems. They typically include a collection of algorithms, data structures, and utilities that can be used to pre-process data, train models, and make predictions on new data. A few of the most known libraries are: *TensorFlow* and *PyTorch*. The latter, or more specifically its C++ API *libtorch*, is our tool of choice for this work.

Tensors

Tensors are the fundamental data structure of many machine learning libraries, including PyTorch. They can be simply thought of as multidimensional arrays, which store and process data efficiently. They are manipulated by specialised tensor operations, which can take advantage of the fact that modern GPUs are highly parallel multi-core processing units that can be used for general-purpose computing (GPGPU).

Automatic Differentiation

Automatic differentiation is a method used in machine learning to efficiently compute the derivatives of functions with respect to their input variables. In most libraries, these can be distinguished by two main modes: *forward mode* and *backward mode*.

In forward mode, the computation is done by tracing the values and derivatives of the function through each operation. In backward mode, the computation is done by tracing the function's output backwards through the operations and computing the derivatives in a reverse order using the chain rule.

In PyTorch, automatic differentiation is called *Autograd*. Any tensor that is created can be marked as `requires_grad`, and from now on every operation on that tensor (and every resulting new tensor), will be tracked and kept in the Computational Graph [8].

2.3 Related Works

There have been various previous works in the field of reproducing images with geometric primitives. One of the most prominent projects is Roger Johansson's *EvoLisa* [9], in which the famous renaissance painting *Mona Lisa* is reconstructed by 50 semi-transparent triangles using a genetic algorithm. Unfortunately, today not much is known about the exact details of Johansson's solution and its performance.

Despite that, Evolisa to this day is a big inspiration for any project in this area, and one of the more recent examples is *Primitive* by Michael Fogleman [10]. Fogleman expands on Johansson's idea and uses various different geometric primitives, which are drawn one by one onto a canvas to minimize the root-mean-square error between pixel values. Instead of using a genetic algorithm, he uses hill climbing as the optimisation method. Given the greedy nature of the algorithm, one would think this is not the best approach for such a problem. However, Fogleman overcomes this by taking advantage of parallel computing to create multiple workers that generate many different starting points, where the best outcome out of all is selected. Fogleman's method is best suited for creating stylised reconstructions. While they properly capture the contents of the original image, the result more closely resembles a water colour painting. Our interest lies in obtaining an approximations that is as close to the reference as possible.

A slightly different approach is taken by Matt Zucker in the *Gabor'2* [11] project. In his solution, the reconstructed image is a sum of 128 Gabor functions, where each parameter was found by using a non-linear least squares solver and Hill Climbing. Additionally, extra constraints on Gabor functions were provided in order to help with reducing high-frequency noise, which they are prone to introduce. Finally, a user-created weight map is introduced to help guide the reconstruction where to exactly the error should be minimised. The final

result is impressively detailed, however the approach of fitting one model at a time on a CPU results in a profound execution time.

> "I let the program run on my 2013 MacBook Pro for a few days to produce the Gabor^2 image...".

mentions Zucker in his personal blog entry. Eventually, the issue of lack of parallelisation was addressed in Zucker's second revision of the project [12], where it was rewritten to run on a GPU by using TensorFlow, which reduced the execution time to approximately 25 minutes. In this updated version the optimisation method is a mix between Adam and Simulated Annealing.

Overall, the discussed projects present various approaches of reproducing images using a sparse amount of primitives. By incorporating the strengths of the previous works, we aim to develop a universal framework for 2D image representation, which can be potentially extended to 3D space.

Chapter 3
Approach

Drawing a geometric primitive onto a raster image typically requires defining values for its translation, rotation, scale and colour. Combining a certain amount of shapes into one canvas can give us an approximation of the reference image. The problem at hand is finding optimal values for all parameters of these shapes. Therefore, the approach to the reconstruction problem can be described by the following steps:

1. Draw a random primitive of a given shape.
2. Calculate the error compared to the reference image.
3. Minimise the error by altering the parameters of the primitive.
4. Repeat until the error is sufficiently small.

Despite the rather simple logic behind it, there are multiple approaches to consider for each step. In this chapter, we go over our method and describe in depth the critical aspects that require particular attention.

3.1 Optimising Algorithm

Gradient Descent based methods are commonly used in machine learning for optimisation due to their efficiency, scalability and the possibility to apply them to a wide range of problems. In recent years we have seen tremendous developments in the field of *Differentiable Rendering*, where the scene is rendered using gradient descent based optimisation methods [13]. By computing the gradients of the rendered image with respect to the scene parameters, it becomes possible to optimise lighting, materials, or even geometry and its location in order to reconstruct or generate a desired scene. The core logic of this technique directly applies to the problem we are trying to solve, hence the choice of Adam as our optimising algorithm.

3.2 Image Error Metric

Calculating the difference between the reconstruction and the reference is the most important part of the whole process. The error holds the information on how close the current result is to the ground truth, and that information is used as a main guiding mechanism for all operations.

The error between two images can be simply expressed as the difference between pixel values. Given grayscale images X and Y of a resolution HxW such error can be calculated using the following methods:

Mean Square Error (MSE) – represents the cumulative squared difference between all pixel values across two images.

$$\mathrm{MSE}(X,Y) = \frac{1}{H}\frac{1}{W}\sum_{y=0}^{H-1}\sum_{x=0}^{W-1}[X(x,y) - Y(x,y)]^2 \tag{3.1}$$

Peak Signal To Noise Ratio (PSNR) – represents the ratio between the maximum possible value of a signal and the power of distorting noise that affects the quality of its representation. R here is the maximum pixel value, which for n-bit image is $2^n - 1$. Because of that parameter and logarithmic scale, it is useful for comparing images of different dynamic ranges but otherwise contains no new information relative to the MSE.

$$\mathrm{PSNR}(X,Y) = 10\log_{10}\left(\frac{R}{\mathrm{MSE}(X,Y)}\right) \tag{3.2}$$

While MSE is a good indication of pixel error, it is not the best representation of human perception when it comes to comparing images. It is possible for MSE values of differently distorted images to be nearly identical, despite great differences in fidelity and quality [14]. Thus we decided to explore different error metrics, which are not solely based on pixel values, and more closely relate to how humans detect differences in images.

Structural Similarity Index Measure (SSIM) – a metric based on the premise that human perception of image quality is directly related to the perceived structural similarity between images, which consists of three components: L - *luminance*, C -*contrast* and S - *structure*. The luminance is determined by the mean of intensity, contrast is determined by the standard deviation of intensity, and the structure is the correlation of the two images [15].

$$L(X,Y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{3.3}$$

$$C(X,Y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{3.4}$$

$$S(X,Y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{3.5}$$

Where:

- μ_x is the mean of pixel values of X;
- μ_y is the mean of pixel values of Y;
- σ_x is the standard deviation of X;
- σ_y is the standard deviation of Y;
- σ_{xy} is the covariance of X and Y;
- C_1, C_2 and C_3 are pre-calculated constants to stabilise the division with weak denominator;

Final SSIM is the weighted combination of these three factors:

$$SSIM(X, Y) = L(X, Y)^{\alpha} \cdot C(X, Y)^{\beta} \cdot S(X, Y)^{\gamma} \tag{3.6}$$

Usually the weights are set to $\alpha = \beta = \gamma = 1$, and $C_3 = \frac{C_2}{2}$. C_1 and C_2 are products of pixel range R and constants, $k_1 = 0.01$ and $k_2 = 0.03$.

$$C_1 = (k_1 R)^2 \tag{3.7}$$

$$C_2 = (k_2 R)^2 \tag{3.8}$$

While one could apply SSIM globally, it is useful to apply it locally and calculate the mean between all the windows, hence the name *Mean SSIM* [15]. It is achieved by using an 11x11 circular-symmetric Gaussian kernel and performing 2D convolution in order to compute local SSIM values.

Regardless of the widespread use and indisputable merits of SSIM, there are edge cases where few of its properties may lead to undefined behaviour, and thus using SSIM as quality assessment measure might lead to incorrect conclusions as well [16]. That is the reason why we also considered a more recent metric, which was created with the goal to address the potential shortcomings of SSIM.

ꟻLIP –a difference evaluator with a particular focus on the differences between rendered images and corresponding ground truths. It outputs a per-pixel difference map between two images, where the differences are approximately proportional to those seen by a human observer when flipping back and forth between the images, located in the same position and without blanking in between [17]. ꟻLIP relies on two parallel pipelines (Fig. 3.1) that calculate the difference between colour and features, which are later combined into a final value. The goal of this approach is to account for the facts that human eyes are more sensitive to chrominance for brighter colours, perceive luminance non-linearly, and they easily pick out edges and any structural discontinuities, especially in pixels.

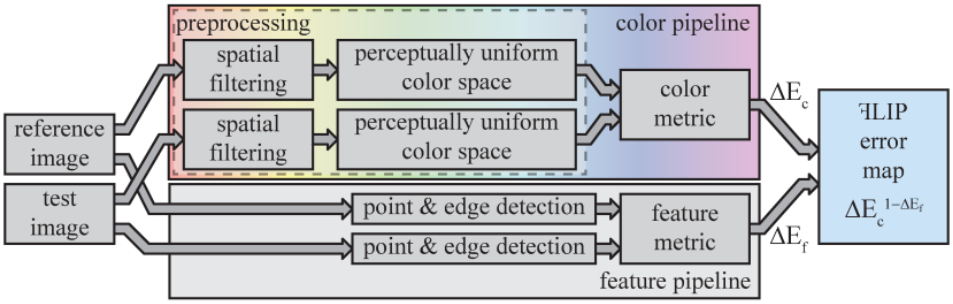


Figure 3.1: The ꟻLIP pipeline [17]

3.3 Additional Constraints

However, just minimising the image error is not sufficient. Quite early in our work it became apparent that some primitives would rather shrink to minimise the error, instead of altering any other parameters. That is why we introduced additional constraints to prevent this behaviour.

$$E(\text{area}) = -min\left(0, 1 - \left(\frac{\text{minArea}}{\text{area}}\right)^2\right) \tag{3.9}$$

The term above returns a penalty if the current area is smaller than the minimal specified area, otherwise returns 0. The idea behind it is to not affect the optimisation if the shape is big enough, but if it shrinks beyond the acceptable value E becomes an increasingly positive value. Minimising it is synonymous with increasing the area of the shape. This constraint is added to the image error and contribute to the final loss value.

3.4 Drawing Primitives

The process of rasterising geometric primitives can not be represented as a differentiable equation, which makes computing the gradients of parameters slightly more complicated. In our initial approach, we worked around this issue by calculating the approximation of the derivative using finite differences. What this mean is, we slightly perturb one of the parameters, draw the updated primitive, then calculate the new error, and finally subtract the original from it and divide by the perturbation value. By repeating the process for each parameter, we can calculate their gradients. However, this approach is incredibly slow, even when we try to fit one primitive at a time, as it needs to be drawn multiple times before the updates are applied. It would be possible to parallelise the algorithm for greater performance, nonetheless, we still would end up using a lot of computing power on multiple renders that do not provide any direct contribution.

Rendering primitives in a way that is fully differentiable holds much greater promise, hence we started exploring other possible solutions that would allow us to represent various geometric primitives. One particular approach that fulfils all the necessary conditions is the adoption of previously mentioned SDFs. Since they represent geometric shapes with fully

differentiable equations, we can take advantage of PyTorch's Autograd. By implementing all the logic with multi-dimensional tensors and moving the computation to the GPU, we end up with completely parallelised gradient calculation and rendering. This not only makes the computation significantly more efficient but also allows us to obtain gradients of all parameters with a single rendering pass thanks to Autograd's backward mode. Additionally, it makes it possible to evaluate multiple primitives at once, each on its separate layer, or even multiple primitives per layer thanks to the fact that SDFs can be combined with Boolean operations.

3.4.1 Blending

Combining these layers is another aspect worth discussing. One possible approach is to use alpha blending to draw semi-transparent shapes onto each other. One drawback of this method is that it is order-dependent, meaning the colour on each successive layer is a direct product of the OVER operator computation with its predecessor. If we want to modify the already processed and drawn primitive, we essentially have to re-render and combine all the preceding layers, apply the modification on the current layer, and re-render all the following layers.

We can avoid this inconvenience by using additive blending, and obtain the final result simply by summing all the layers together. By definition, addition is an order-independent operation, thus the layers can be combined in any order and still produce the same image. Furthermore, there is no consideration of the alpha channel, which lets us reduce the colour information to be just stored as RGB values.

3.4.2 Types of Primitives

We use 3 types of base geometric shapes to represent with SDFs, Circles, Boxes and Triangles (Fig. 3.2). Additionally, we consider the Circle displaced with two convolved sine waves, where their frequency is controlled by the current coordinates multiplied by a displacement factor (Fig. 3.3). We choose to evaluate only the displaced Circles, since the results of such displacement in 2D yield similar results for all considered base primitives, and the computational complexity of the Circle is the lowest.

(a) (b) (c)

Figure 3.2: Types of used base SDF primitives

(a) 0.7 **(b)** 2.0 **(c)** 10.0

Figure 3.3: Visualisation of the Circle SDF displaced with 2 convolved sine waves of varying displaced factors

Chapter 4
Implementation

The reconstruction can be thought of as a closed loop (Fig. 4.1), in which we aim to minimise the loss function, which is described by one of the image error metrics and additional constraints for geometric shapes. By calculating the gradients with respect to primitives' parameters and using gradient descent, we are able to adjust these parameters in order to reduce the difference between the reconstruction and the reference. By iterating this process we try to determine the best possible state for all primitives, which is the optimal approximation of the original image.

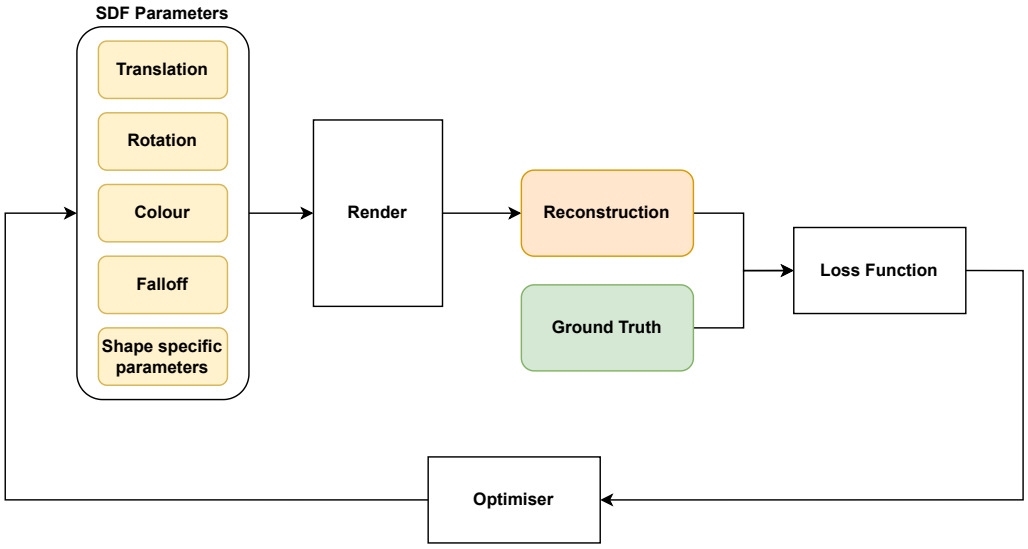


Figure 4.1: A block diagram describing the reconstruction loop

In this chapter, we describe the steps of the reconstruction process in greater detail and go over how it was implemented. The project's application code is written in C++, and PyTorch API is used for solving the optimisation problem and rendering SDFs.

4.1 Image Storage

The first important step is to read the reference image to memory, and then transfer the pixel data into a 4D PyTorch tensor of shape $N \times C \times H \times W$, or just $NCHW$ for short, where:

- N is the number of independent layers,
- C is the number of colour channels in the image,
- H is the height of the image,
- W is the width of the image.

Given an RGB image with resolution 640×480 it will be stored as a $1 \times 3 \times 480 \times 640$ tensor.

Once the data is converted into a tensor, we set its type to float and normalise the colour values. The original is kept around to be used as the source for performing bilinear interpolation to produce a scaled down version of the reference.

4.2 Pixel Coordinates

Once the image data is properly stored, we create a tensor of pixel coordinates based on the resolution of the reference. Since throughout the reconstruction process we are operating at different resolutions, these coordinates are normalised along the shortest side (Fig 4.2). This way we avoid non-uniform scaling, which would otherwise introduce distortions. This also ensures that the primitive drawn in the down scaled image will be placed at approximately the same position in the full scale image without the need to remap its origin. Whenever the resolution of the reference changes, the pixel coordinates are recalculated.

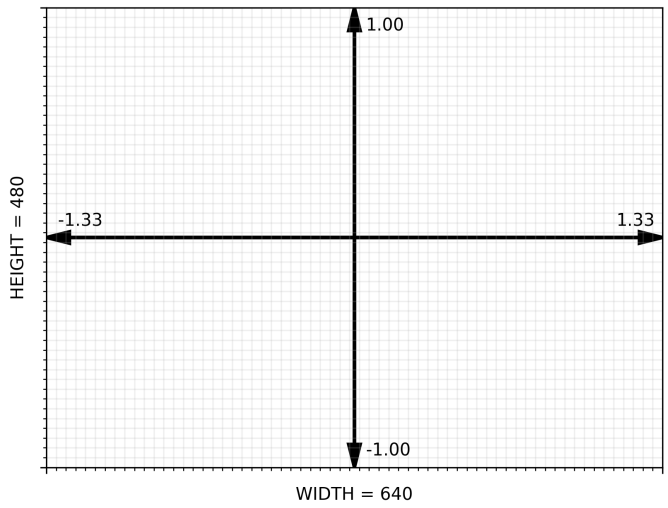


Figure 4.2: Normalised pixel coordinates

4.3 Primitive Parameters

Next, we initialise the list of trainable parameters for the geometric primitives. We start with common parameters, which are shared between all types of shapes, which are:

Translation - controls the position of the primitive on the image. Initially set to a random position in the bounds of normalised pixel coordinates.

Rotation - controls the rotation of the primitive. Initially set to 0.

Colour - controls the RGB colour value of the primitive. Since we are using additive blending to mix the layers together it can be in the range $(-1.0, 1.0)$. Initially set to 0.

Falloff - controls the falloff of colour intensity from the center of the primitive. Clamped to be always in the range $(0.01, 0.99)$, where minimum value means a sharp anti-aliased shape, and maximum value results in an extremely blurred shape. Initially set to 0.5.

To unify the data representation and streamline the multidimensional operations, parameters are also created as tensors of shape $NCHW$, where:

- N represents the number of layers,
- C depends on parameter data type (3 for RGB, 2 for translation, 1 for rotation and falloff),
- H and W are simply set to 1.

For example, if we are to evaluate 64 primitives, then the shapes of parameter tensors in the list will look like the following:

- **Translation** - $64 \times 2 \times 1 \times 1$
- **Rotation** - $64 \times 1 \times 1 \times 1$
- **Colour** - $64 \times 3 \times 1 \times 1$
- **Falloff** - $64 \times 1 \times 1 \times 1$

Shape-specific parameters, e.g. radius for a circle, are created accordingly and inserted at the end of the list.

4.4 Evaluating SDFs

We have already described what SDFs are, and why they are useful in solving the task at hand, now let us have a closer look at how they are evaluated into 2D geometric shapes, using the SDF for a circle as an example.

sdCricle (Lst. 1) takes two arguments, *p* - current point, *r* - radius size, and returns distance from the point to the surface of the circle. This distance is used to determine colours on the rendered image. If the distance is positive the pixel is set to the background colour, otherwise it is set to the desired colour of the shape (Lst.2).

```
sdCircle(p, r)
{
    return length(p)-r;
}
```

Listing 1: Circle SDF [18]

```
d = sdCircle(p, r);
col = d > 0 ? bgCol : shapeCol;
```

Listing 2: Coloured circle SDF [18]

This will however result in a rather pixelated circle, especially for lower resolutions. The distance can also be used to apply anti-aliasing to produce smooth edges. Instead of using the logical operation to determine the colour, we can interpolate between the background and the shape colour using *smoothstep* of the distance as the weight (Lst. 3).

```
col = lerp(bgCol, shapeCol, 1.0 - smoothstep(0.0, 0.01, d));
```

Listing 3: Coloured circle SDF (anti-aliased) [18]

Increasing the upper bound of the smoothstep term increases the band of the smoothing operation, which allows us to achieve an effect that resembles the falloff of colour intensity from the surface (Fig. 4.3).

(a) Falloff = 0.01 **(b)** Falloff = 0.99

Figure 4.3: Circles of equal radius with different falloff values

SDFs are always drawn at the origin, meaning the evaluated point needs to be transformed to get a translated, rotated and scaled object. This can be easily achieved by multiplying the point with the inverse of a transformation matrix. However, the non-uniform scaling compresses or dilates space, therefore the returned distance is not exact. Since we only care

about the visualisation of the shape this aspect can be potentially ignored. Nevertheless, we chose to keep the exact representation, and control the size of primitives by their specific parameters, in this case, the radius.

By rendering the primitive with SDFs we ensure that each pixel value in the resulting image is a direct result of the fully differentiable equation. However, some information about the gradient can be lost due to the definition of the smoothstep, as it needs to normalise the input value to the given bounds and clamp it to range 0 to 1. If the normalised value is not in the clamp range the resulting derivative is 0, which causes the whole gradient computation using chain rule to be 0. To avoid this behaviour an altered *sigmoid* (Eq. 4.1) can be used as an alternative smoothing function. Sigmoid, similarly to smoothstep, is used to map the value to the range 0 to 1. Although, in its base form, the curve is much wider and centred around Y axis (Fig. 4.4a). Altering the input to be $10x - 5$ normalises the curve (Fig. 4.4b) [19].

$$S(x) = \frac{1}{1 + e^{(-x)}} \tag{4.1}$$

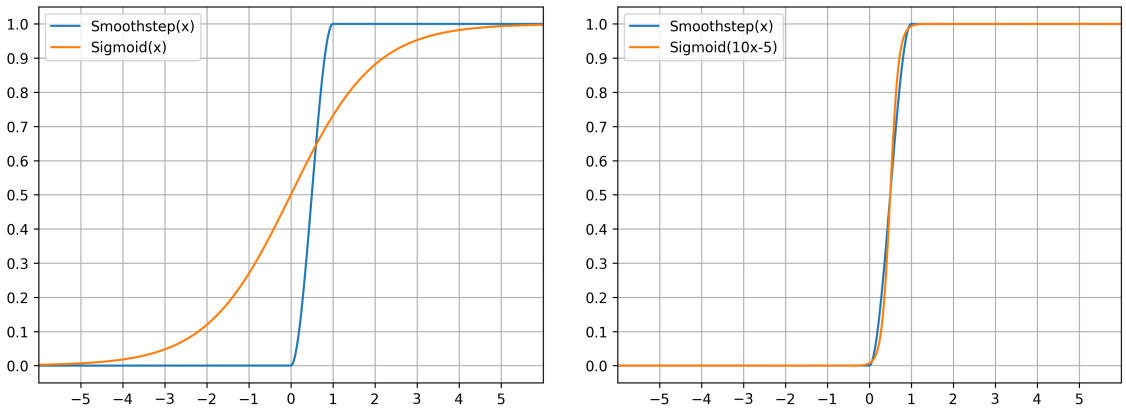


Figure 4.4: The comparison of base (a) and normalised (b) sigmoid with smoothstep

4.5 Layers Composition

Passing parameters in $NCHW$ format to SDF implemented with PyTorch makes it possible to evaluate in parallel N different primitives, resulting in N rendered layers of individual primitives, which are summed together to create the reconstructed image (Fig. 4.5).

Alternatively, it is possible to specify the number of shapes per layer, which will be then blended using *smooth minimum* operation. The union of SDFs is achieved by taking the minimum of the returned distance between two functions. Smooth minimum is a special case of the union, which behaves like the base one for the areas where the primitives are further apart, but creates smooth transitions where their distance values are close to each other. Additionally, it can be used to compute the mix factor which is used for the smooth blending of the colours of the primitives (Lst. 4). There are numerous methods to accomplish this behaviour, but our interest lies specifically in the exponential version of the smooth minimum,

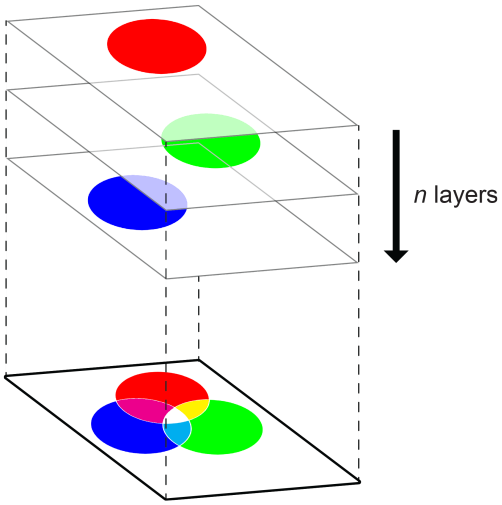


Figure 4.5: Obtaining the reconstructed image by composition of N layers of evaluated SDFs

as it is able to generalise more than two distances and produce the same results regardless of the order of the operations [20]. It is derived from *LogSumExp*, which smoothly approximates the maximum function by calculating a logarithm of the sum of the exponentials of the arguments (Eq. 4.2). Introducing a negative factor transforms it into an approximation of the minimum function (Eq. 4.3). In the context of blending SDFs, k can be used to control the range of the smooth union.

$$SMAX(x) = LSE(x) = log\left(\sum_{i=0}^{N-1} e^{x_i}\right) \tag{4.2}$$

$$SMIN(x) = -\frac{1}{k}LSE(x) = log\left(\sum_{i=1}^{N-1} e^{-kx_i}\right) \tag{4.3}$$

```
smin(d1, d2, k = 10.0)
{
    f1 = exp2(-k, d1);
    f2 = exp2(-k, d2);

    return -log2(f1+f2)/k , f2;
}
```

Listing 4: Example of the exponential smooth minimum, which returns the minimum distance and the mix factor

To achieve order-independent smooth colour mixing, we incorporate the weighted additive blending [21], and use the mix factor as a weight of individual colours of primitives. The

final colour is then simply the weighted average of all contributing colours (Eq. 4.4). If the primitives create an union, then their colours are blended, otherwise, they keep their original colour.

$$C = \frac{\sum_{i=0}^{N-1} C_i w_i}{\sum_{i=0}^{N-1} w_i} \tag{4.4}$$

Figure 4.6: A smooth union between a circle and a box of different colours

4.6 Reconstruction Phases

Now that the setup process and the rendering methods are explained, let us carry on with the description of the reconstruction algorithm. We found that fitting primitives globally, all at the same time, tends to yield much more promising results, than if the learning were done in batches. Unfortunately, working with a number of primitives greater than 128 and images of resolution bigger than 512×512 takes an immense toll on memory usage. After all, in this case, we render 128 layers, each with 262,144 pixel values, which in total is 33,554,432 pixels. Moreover, PyTorch's Adam optimiser stores a list of all the operations that contribute to each pixel value. For that reason the reconstruction is done in two phases:

Global fitting - the working space is limited to 128×128 resolution (or equivalent, keeping the aspect ratio of the original image). This low resolution preserves the most important features and makes it possible to fit a massive amount of primitives at once. Run for approximately 500-600 steps.

Batch re-fitting - once the initial fitting at low resolution is done, the working space is up-scaled to half of the original resolution, and the primitives are divided into equal batches, which are further optimised given the new details of higher resolution. The inactive batches are combined and saved to a static tensor, which is added to the evaluation of the active batch to produce the reconstructed image. Each batch runs for approximately 150-250 steps.

Finally, the working space is set to the original resolution, and the primitives are drawn batch by batch to produce the final results.

We also tried to randomly re-initialise small batches after the re-fit phase in order to try to escape from potential local minima, fit the new set of parameters, and accept if it improves the overall quality, or discard otherwise. However, better states of parameters for a batch were rarely found, and if there was any improvement, it was negligible given the amount of work it requires.

Chapter 5
Results

In this chapter, we present and evaluate different modes of implementation. This includes comparison of images obtained with implemented metrics, for varying amounts and different types of primitives. All results were rendered using PyTorch (CUDA 11.7).

The final implementation includes MSE and SSIM as image error functions. Unfortunately, we were unable to use ꟻLIP as a learning loss, because the returned gradients were always 0. Most likely due to the fact the algorithm requires many different *clamp* operations, which cause the gradient information to be lost, as mentioned in Section 4.4.

However, we still use ꟻLIP as an objective method to evaluate the results obtained both with MSE and SSIM, as those two metrics operate in different ranges and should not be directly compared. The final ꟻLIP score is in range [0, 1], and is obtained by taking the mean of the difference values. The lesser the score the closer the tested image is to the reference. Below are presented the reconstructions (Fig. 5.2) of the reference image (Fig. 5.1), recreated with 256 primitives of all types, for both MSE and SSIM. Results for 64, 128, 512 and 1024 shapes can be found in the appendix A.

Figure 5.1: The reference image - Woman Dark Hair 512×512 [22]

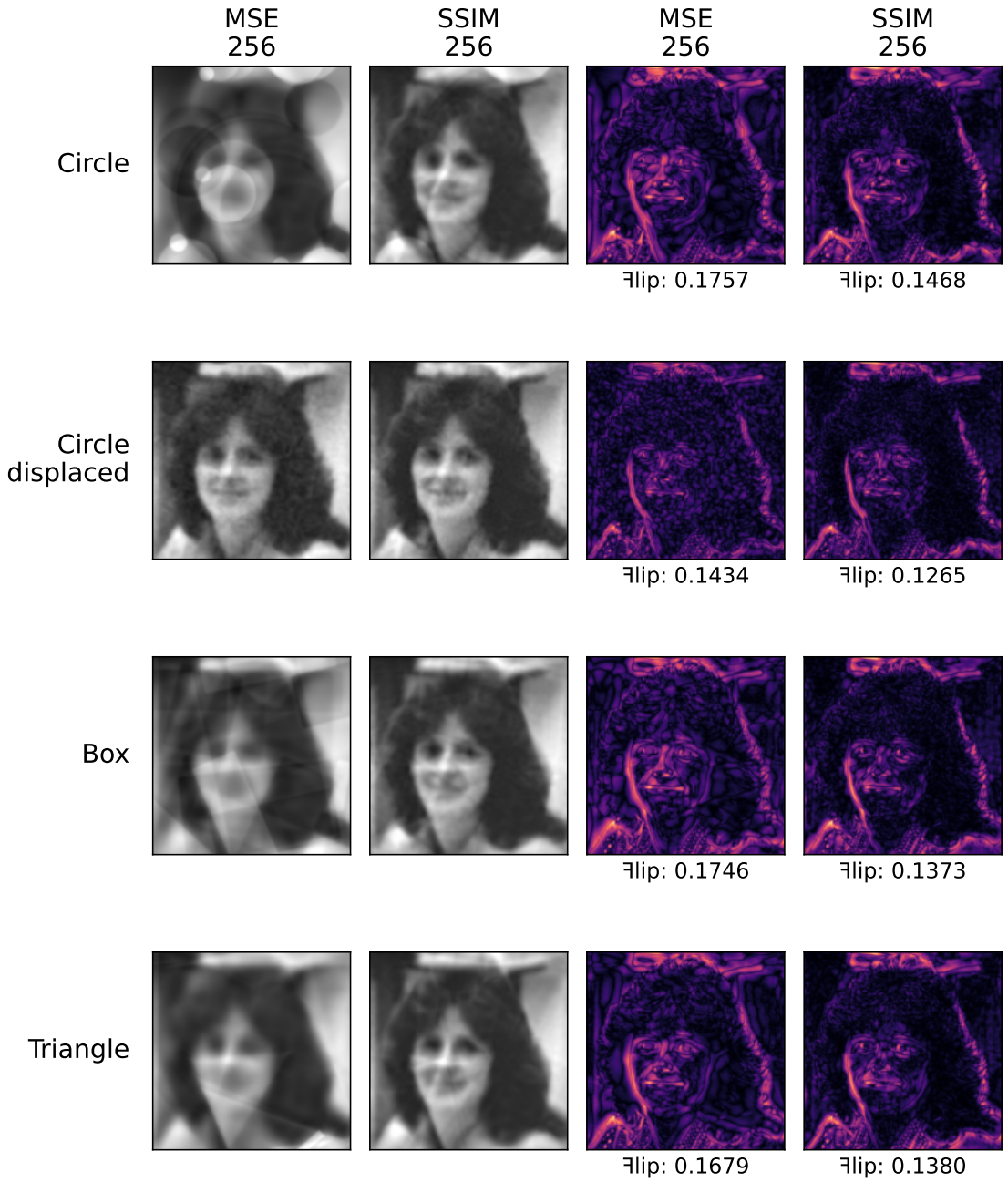


Figure 5.2: Comparison of results obtained using MSE and SSIM for 256 primitives and their difference maps

N primitives	Error Metric	SDF Type			
		Circle	Circle (Displaced)	Box	Triangle
64	MSE	0.2122	0.1825	0.1948	0.1912
	SSIM	0.2186	0.1852	0.1870	0.1721
128	MSE	0.2077	0.1674	0.1879	0.1811
	SSIM	0.1776	0.1595	0.1537	0.1612
256	MSE	0.1757	0.1434	0.1746	0.1679
	SSIM	0.1468	0.1265	0.1373	0.1380
512	MSE	0.1434	0.1081	0.1343	0.1330
	SSIM	0.1190	0.0949	0.1060	0.1075
1024	MSE	0.1372	0.0918	0.1277	0.1243
	SSIM	0.1093	0.0781	0.0993	0.0972

Table 5.1: Final scores for results obtained with MSE and SSIM

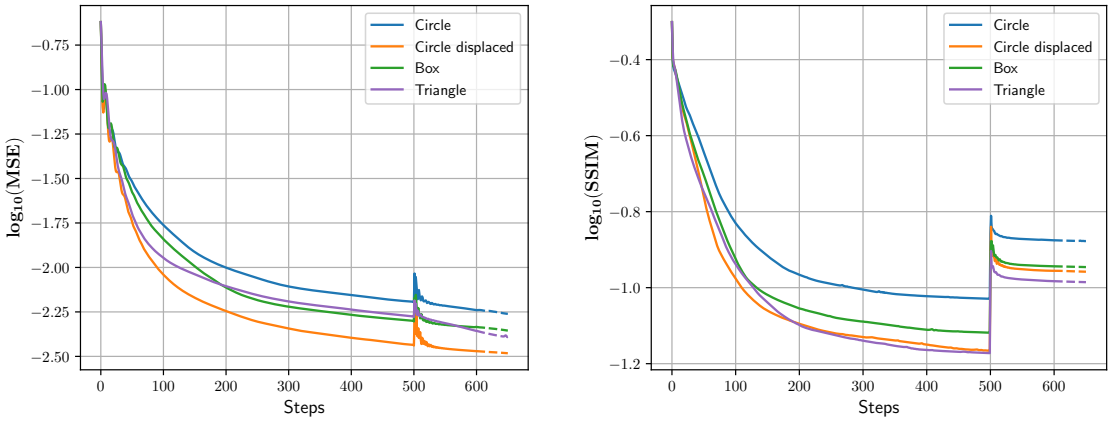
5.1 Error Metrics

Let us take a look at the final scores of the reconstructions with different configurations (Tab. 5.1). For a lower number of primitives, there is a minor difference in perceived quality, and between the ꟻLIP scores of the two metrics. However, for 128 and more that difference gradually grows, and it becomes quite clear that SSIM performs better. Looking at produced images (Fig. A.2 - A.6) and their difference maps, we can see that MSE tends to distribute the error across the whole image, which results in a much noisier approximation. SSIM is highly sensitive to any inconsistencies in the structural information, which is best shown during the learning process. The total SSIM value, used as a main component of the loss, rapidly decreases during the initial-low resolution phase (*solid line*), and then visibly rises when the new information is introduced after the increase in the resolution for the batch refitting phase (*dashed line*) (Fig. 5.3b). Whereas, for MSE this rise is much less significant (Fig. 5.3a). That being established, all the results that are discussed in the following sections relate to the reconstructions obtained using SSIM.

5.2 Primitives

Comparing the results and their scores for different primitives it is apparent that the base Circle performs the worst, and it requires a great number of them to produce a satisfying reconstruction. The quality of the Box and the Triangle is notably better, and their results are comparable to each other for almost all configurations. The Displaced Circle produces the best results.

64 primitives are enough to capture the general structure of the reference, but not sufficient to produce more detailed features, in this case, eyes, nose and mouth (Fig. A.2), although for the Displaced Circle, they become distinguishable. There is a notable improvement with each increase in the number of primitives. At 256 (Fig. 5.2) all the critical features are notable, and the results start to resemble a blurred version of the original. There are still visible



(a) (b)

Figure 5.3: MSE (a) and SSIM (b) values during learning

artefacts in the form of sharp edges of the primitives, which are the side effect of additive blending. At 512 (Fig. A.5) the image is less blurry, and for the Displaced Circle, even small details become distinct, like eyebrows, eye whites, and under-eye wrinkles. For 1024 (Fig. A.6) the result is even sharper, and again the Displaced Circle is able to recreate even minor details as single hair curls, or the chequered texture of the collar.

5.2.1 Smooth union

Now let us evaluate reconstructions composed of smooth unions of SDFs of the same type compared to the individual primitives. Figure 5.4 contains the results, where on each layer the smooth minimum of 4 primitives was calculated to create a smooth union, and Table 5.2 shows the final ꟻLIP scores. While this approach allows for a smoother transition in colour, most of the sharp edges are lost due the shapes being blended together. For a lower number of primitives, the outcome resembles a smudged image. After all, if each layer contains a union of 4 shapes, this can be thought of as $N/4$ unique primitives rather than N. Overall, differences in ꟻLIP scores between the individual and joined primitives are minor for 512 and lower, and only for 1024 the improvement is notable, nevertheless only the smooth union of Triangles achieves results comparable to the Displaced Circle.

		SDF Type						
N primitives	Metric	Circle	Circle (smin)	Box	Box (smin)	Triangle	Triangle (smin)	Circle (Disp.)
128	SSIM	0.1776	0.1881	0.1537	0.1653	0.1612	0.1735	0.1595
256	SSIM	0.1468	0.1522	0.1373	0.1305	0.1380	0.1348	0.1265
512	SSIM	0.1190	0.1107	0.1060	0.1047	0.1075	0.0995	0.0949
1024	SSIM	0.1093	0.0945	0.0993	0.0815	0.0972	0.0784	0.0781

Table 5.2: Final scores for results of all types of primitives

Figure 5.4: Comparison of results for smooth union of 4 primitives per layer

5.3 Additional Tests

So far for the test reference, we have been using a grayscale image with a big object occupying most of the frame. In this section we present additional results (Fig. 5.6) for images with more varied contents (Fig. 5.5) for the most promising configurations of the reconstruction, using SSIM as the image error metric, and the Displaced Circle as the primitive type.

As already established, our solution can efficiently capture big objects that have a significant change in contrast compared to the background (Fig. 5.5a, 5.5d). However, if the scene has many complex smaller objects located far from the camera, it struggles to properly represent those for a smaller amount of primitives, like the buildings behind the cameraman (Fig 5.5a), or the living room furniture (Fig. 5.5b). Unfortunately, images of low contrast, even if in colour, tend to be the hardest to reproduce, unless a great amount of primitives is used (Fig. 5.5c).

(a) (b) (c) (d)

Figure 5.5: Additional reference images **512 × 512** [22] - Cameraman (a), Living room (b), F16 (c), Peppers (d)

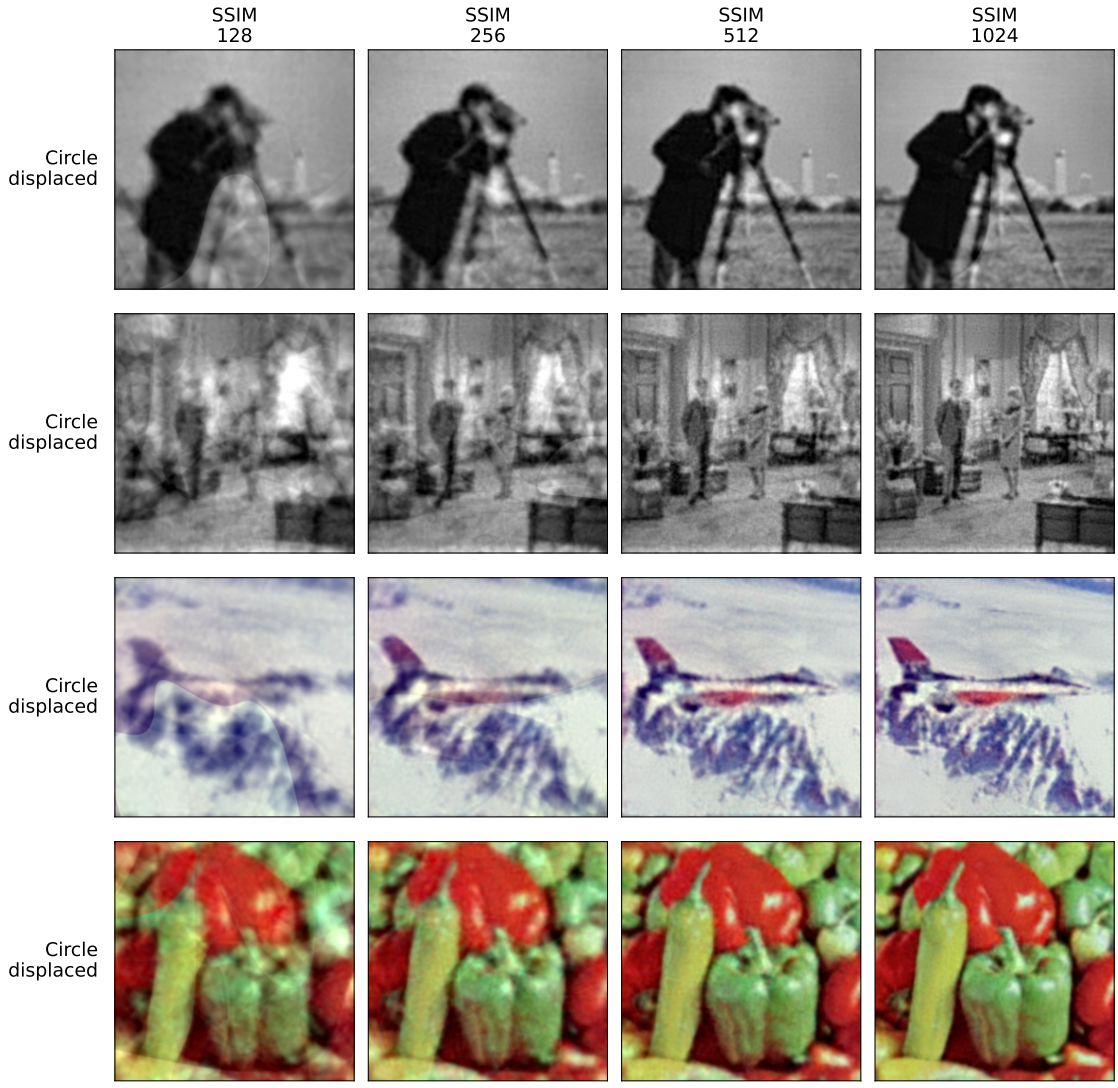
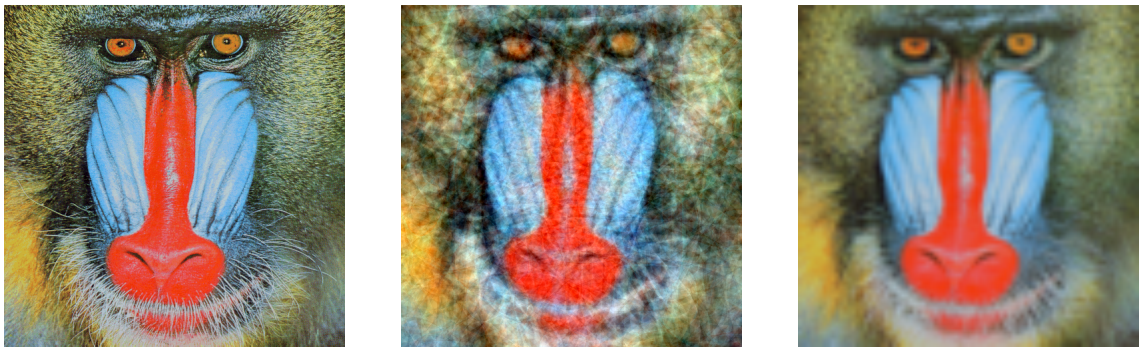


Figure 5.6: Additional images - Reconstructions

5.4 Raster images

Finally, let us discuss how our approach compares to raster images with sparse pixel data. Figure 5.7 contains side by side comparison of the original image, the reconstruction with 512 Displaced Circles, and its scaled down version of 83×83 resolution. To represent the image with 512 Displaced Circles, where each primitive is described by 10 values of 32 bits each [1], requires 163,840 bits. Whereas, the original uncompressed RGB raster image of resolution 512×512, requires 262,144 pixels, each storing 3 colour channels of 8 bits, therefore 6,291,456 bits. A raster image of equivalent size of the reconstruction is of 83×83 resolution, 165,336 bits in total.

Although the reconstruction is not a perfect copy of the original, it is able to encompass great details with just a fraction of the size of the reference. Despite the fact that the final result contains noticeable noise, it is not as blurry as the low scaled version of the original. Additionally, possibility to recalculate the primitives described by SDFs potentially allows for performing scaling operations without the loss of quality, similar to vector graphics.



(a) **(b)** **(c)**

Figure 5.7: Comparison of images represented with equivalent amount of pixels and geometric primitives: Original 512×512 (a), 512 Displaced Circles (b), Original 83×83 (c)

[1] Translation: 2, Rotation: 1, Colour: 3, Falloff: 1, Radius: 1, Displacement Factor: 2

Chapter 6

Conclusion

In conclusion, in this thesis we explored an alternative approach to represent digital images using differentiable rendering based on Signed Distance Functions. Throughout our investigation, we evaluated various methods that hold potential for this task, and thoroughly discussed the most promising configurations.

In section 5.1 we conduct a comparison between the results obtained with two different error metrics. The goal of this analysis aims to determine which metric is more suitable for solving such problems. Reconstructions obtained with SSIM are notably of higher quality, indicating that perceptual error metrics should be considered if the goal is to achieve detailed, and faithful reconstructions. Section 5.2 discusses the efficiency of different implemented types of primitives. We assess their ability to describe natural shapes of image contents and examined how the quality of the reconstruction is influenced by the number of geometric primitives. In most cases, approximately 64 primitives proved to be sufficient to capture the general structure of the image. Employing 256 or more primitives results in increasingly more detailed outcomes. Moreover, we demonstrate that possibility of creating new complex shapes by combining base SDFs using Boolean Operation is also a viable option and yields promising results for greater number of primitives. In section 5.3 we present additional results for images of different types and contents, and address the shortcomings of our approach. While it is able to properly recreate big objects in the foreground, it struggles with background scenery and low-contrast changes. Finally, we drew a direct comparison between the representation of images in our proposed alternative approach and the traditional raster image representation. By doing so, we highlighted the potential advantages offered by our proposed methodology.

Overall the presented results show that it is possible to obtain an alternative representation of the digital image by leveraging the principles of differentiable rendering and utilising geometric primitives described by SDFs.

6.1 Limitations

One aspect that we consider a limitation is the efficiency of how SDFs are obtained. The current approach evaluates the distance for every pixel, which as shown can be not only computationally demanding, but on top of that requires a lot of memory. The need to evaluate every possible coordinate in the space leads to unnecessary calculations for very small shapes. Much more optimal approach would be to evaluate only the pixels that are bound by its Axis Aligned Bounding Box. However, then the resulting tensors of the evaluated SDFs would differ in their dimensions, and due to the requirements of parallel computation in PyTorch, tensors need to be of the same dimensions. For the same reason we were unable to explore fitting multiple types of primitives at the same time, which could potentially lead to better shape approximation.

Finding alternative strategies or frameworks that allow for such parallel computation with varying tensor dimensions could enhance the evaluation of SDFs and improve the overall efficiency of our method.

6.2 Future Work

While our project presents a successful approach to image reconstruction using 2D geometric primitives rendered with SDFs, there are areas that could benefit from further improvement in future works. One significant aspect for enhancement is the refinement of the algorithm to achieve higher-quality results with a smaller number of primitives. Currently, our approach produces satisfactory outcomes, but optimising the algorithm to reconstruct images with fewer primitives would not only improve computational efficiency but also allow for more concise representations of the reconstructed images.

Another area worth exploring is the alternative way of compositing the layers together. Instead of additive blending it would be interesting to explore different methods to fully utilise SDF's ability to be combined with Boolean operations, which potentially could lead to improved shape representation. While we touched upon possibility of such solution by implementing smooth unions of primitives, extending it to take advantage of intersections and subtractions requires an approach more reminiscent of decision trees, which are not exactly differentiable.

Furthermore, the most exciting possibility is the extensions of our approach beyond the reconstruction of 2D images. Given that SDFs can be used for representation of 3D objects, the core concepts of this project can be applied to 3D model reconstruction. By either sampling the reference data from renders depicting the model from different angles, or utilising point clouds as ground truth.

References

[1] Wikipedia contributors. Constructive solid geometry, April 2023. https://en.wikipedia.org/w/index.php?title=Constructive_solid_geometry&oldid=1149352740.

[2] B. Esfahbood. GLyphy is a signed-distance-field (SDF) text renderer using OpenGL ES2 shading language., 2014. https://github.com/behdad/glyphy.

[3] A. Scandurra. Leveraging Rust and the GPU to render user interfaces at 120 FPS, March 2023. https://zed.dev/blog/videogame.

[4] I. Quilez. Painting a Character with Maths, November 2020. https://youtu.be/8--5LwHRhjk.

[5] GODOT Docs. Signed distance field global illumination (SDFGI), June 2020. https://docs.godotengine.org/en/stable/tutorials/3d/global_illumination/using_sdfgi.html.

[6] S. Ruder. An overview of gradient descent optimization algorithms. 2016.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

[8] PyTorch. A Gentle Introduction to torch.autograd, 2023. https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html.

[9] R. Johansson. Genetic Programming: Evolution of Mona Lisa, December 2008. https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/.

[10] M. Fogleman. Primitive: Reproducing images with geometric primitives., September 2016. https://github.com/fogleman/primitive.

[11] M. Zucker. Gabor^2, August 2016. https://mzucker.github.io/2016/08/01/gabor-2.html.

[12] M. Zucker. Image fitting TensorFlow rewrite, April 2018. https://mzucker.github.io/2018/04/27/image-fitting-tensorflow-rewrite.html.

[13] H. Kato, D. Beker, M. Morariu, T. Ando, T. Matsuoka, W. Kehl, and A. Gaidon. Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057*, 2020.

[14] Wang Zhou and A.C. Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE Signal Processing Magazine, Signal Processing Magazine, IEEE, IEEE Signal Process. Mag*, 26(1):98 – 117, 2009.

[15] Wang Zhou, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing, Image Processing, IEEE Transactions on, IEEE Trans. on Image Process*, 13(4):600 – 612, 2004.

[16] J. Nilsson and T. Akenine-Möller. Understanding ssim. 2020.

[17] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, and M.D. Fairchild. ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020.

[18] I. Quilez. Disk - distance 2D, February 2020. https://www.shadertoy.com/view/3ltSW2.

[19] tholzer. Smooth Step Alternatives, October 2017. https://www.shadertoy.com/view/ltjcWW.

[20] I. Quilez. Smooth minimum. https://iquilezles.org/articles/smin/.

[21] I. Quilez. Color blending exp-smooth-min, August 2021. https://youtu.be/8--5LwHRhjk.

[22] USC-SIPI. The USC-SIPI Image Database, 1977. https://sipi.usc.edu/database/.

Appendices

Appendix A
Images

Figure A.1: The reference image - Woman Dark Hair **512 × 512**

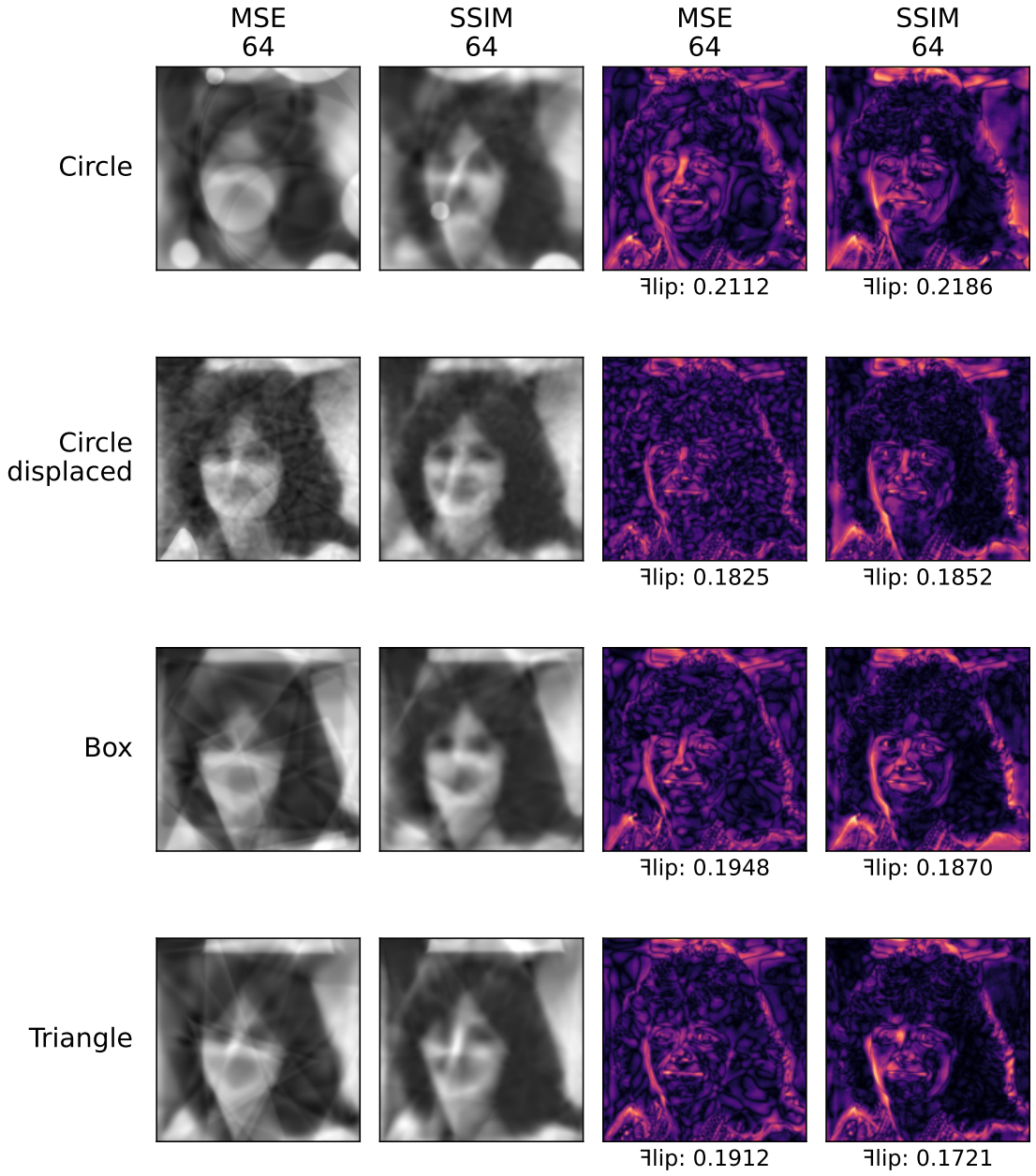


Figure A.2: Comparison of results obtained using MSE and SSIM for 64 primitives and their difference maps

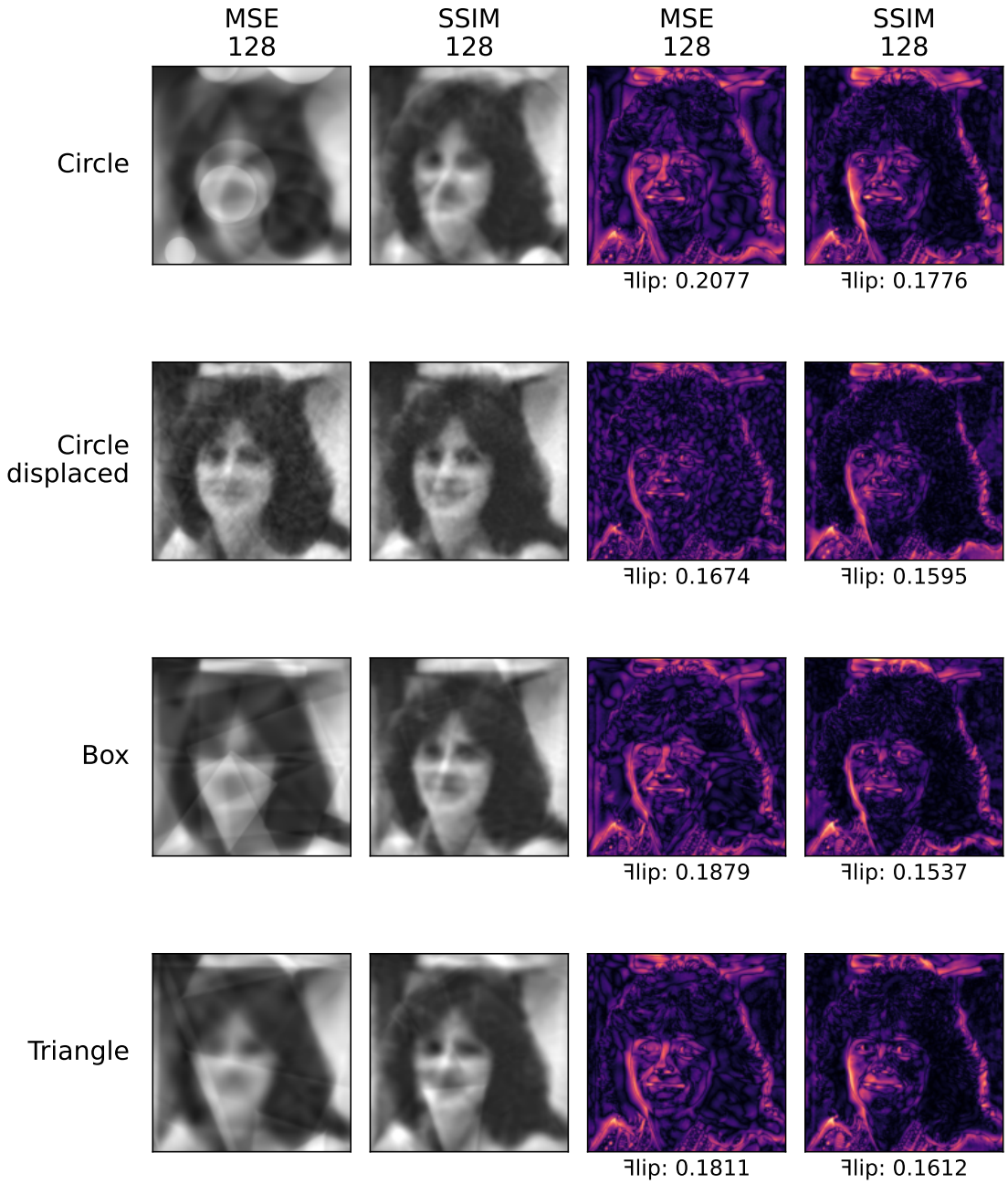


Figure A.3: Comparison of results obtained using MSE and SSIM for 128 primitives and their difference maps

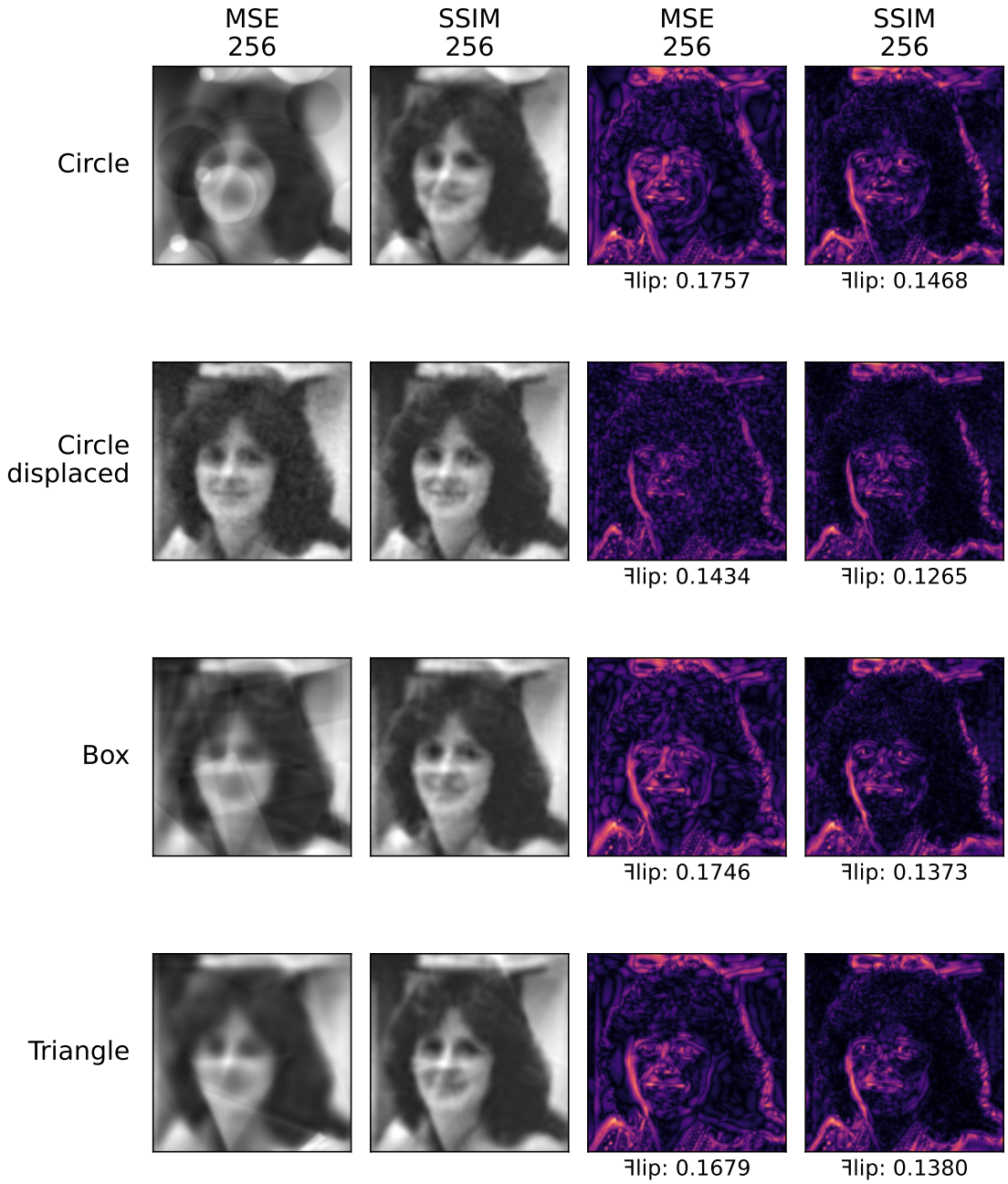


Figure A.4: Comparison of results obtained using MSE and SSIM for 256 primitives and their difference maps

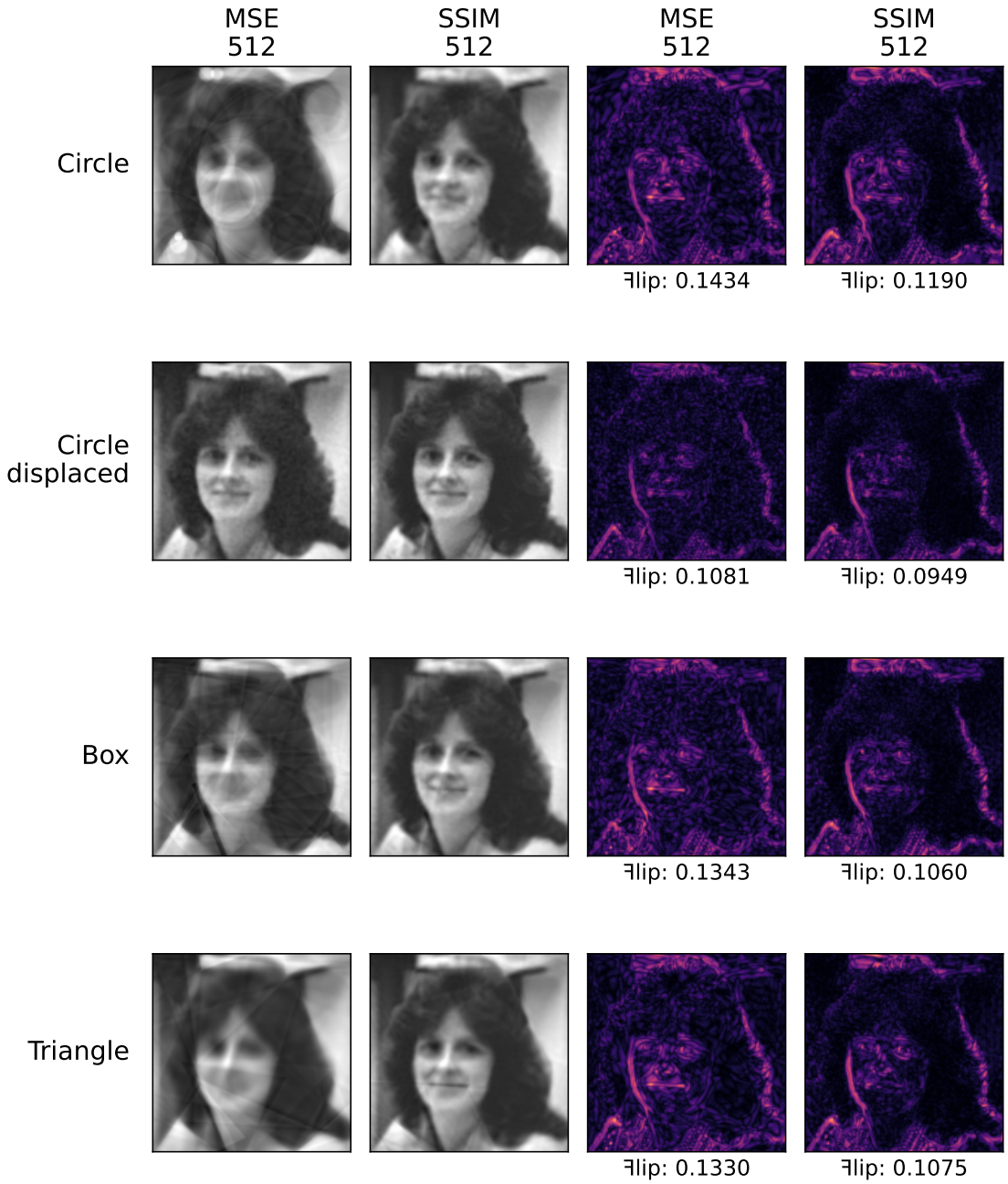


Figure A.5: Comparison of results obtained using MSE and SSIM for 512 primitives and their difference maps

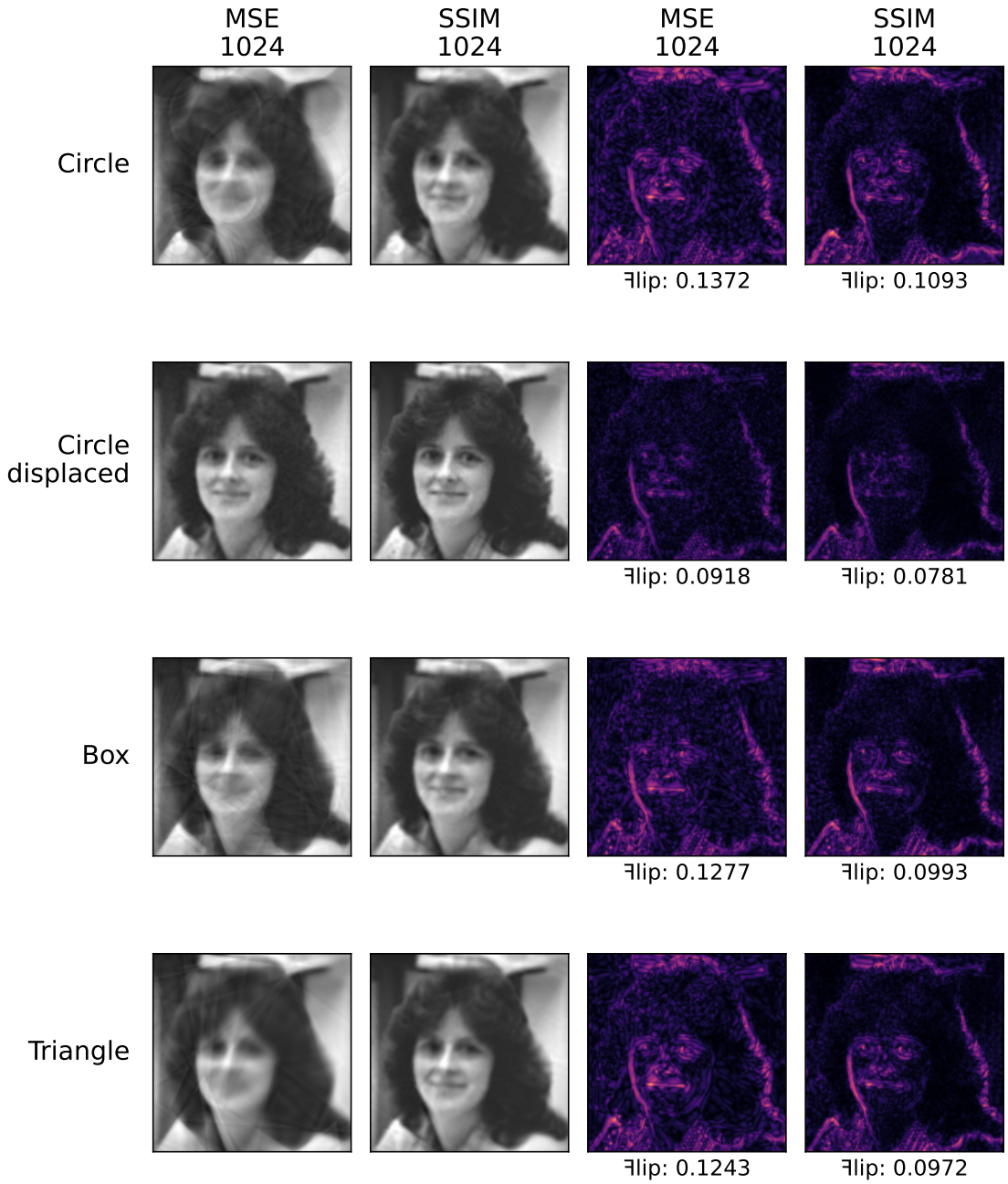


Figure A.6: Comparison of results obtained using MSE and SSIM for 1024 primitives and their difference maps