Illusion3D: 3D Multiview Illusion with 2D Diffusion Priors

Spencer Lutz

May 2025

1 Introduction

A multiview illusion is a known artistic technique in which a single work can present multiple diverse interpretations when viewed from different perspectives. A couple common examples are the old lady vs. young woman drawing, or the common duck vs. rabbit idea (Figure 1). In recent years, machine learning has been used to generate multiview illusion artworks in various media, like 2D art [1, 3], wire art [4], or sculpture [5].

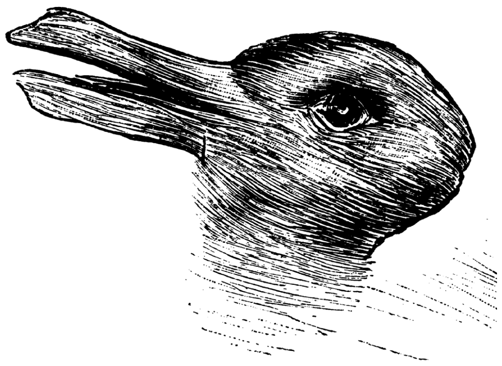


Figure 1: One of the earliest and most popular examples of a multiview illusion, this image appears like a duck when turned one way and a rabbit when turned another.

Our work, Illusion3D: 3D Multiview Illusion with 2D Diffusion Priors [2] takes the concept of a multiview illusion and extends it to various scenes in 3D. We use diffusion guided by natural language prompts to generate textures and shapes for objects such that they take on different appearances when viewed in different ways. We cover four setups: a cube, of which we vary the set of faces that are viewed; a sphere, of which we vary the angle at which it is viewed; a 2D surface, on which we place one or more reflective objects; and a Rubik's cube, which we shuffle to reveal multiple sets of images.

Our basic approach to this problem is to render the object from all views of interest, run a single diffusion step on each view, and average the results. Over many iterations, this allows the model to converge upon a texture that aligns with the given prompt for each view. We then build from this basic approach with a set of novel techniques that substantially improve the quality of results.

My primary contribution to this research project was the differentiable rendering and training framework. Each of these cases required unique approaches to representing the objects of interest in a way that allows us to apply diffusion. The overarching requirement was the ability to take as input a 3D model and proposed texture, and output a render of that 3D model with the texture applied, in such a way that we can compute the gradient of a rendered pixel with respect to a pixel of the texture. This differentiability is necessary for the diffusion process to be able to generate a texture based on an output render. For much of this portion, I also worked alongside my co-author Vaibhav Sanjay. The goal of our contributions was to set everything up so that other researchers could focus on improving diffusion techniques without having to worry about any of the behind-the-scenes scaffolding.

2 Background

2.1 Differentiable Rendering

Diffusion works by iteratively removing noise from a random image until a desired image is produced. Individual pixels in the image are updated based on this learned de-noising. To apply this to the case of 2-dimensional textures rendered in 3-dimensional views, we cannot simply change pixels in the output 3D render. This is because we are learning the 2D texture which, after heavy transformation, only makes up a part of the output render.

To perform the de-noising in this setup, we need to know how each pixel in the output render is affected by the pixels in the texture we're optimizing. This means our renderer cannot be a black box that simply takes in a scene and outputs a render. We need to be able to directly track how changes in the texture lead to changes in the render – this brings to mind the idea of a gradient. If we can compute the gradient of the rendered pixels with respect to the texture pixels (a Jacobian), we can reverse that mapping when updating pixels in the scene and appropriately de-noise the texture according to the render.

Luckily, PyTorch3D [6], the rendering library we used, handles much of this for us. All we need to do is ensure that our rendering is *differentiable*, and then PyTorch3D handles the rest. For our renderer to be differentiable, we must use built-in PyTorch3D operations and functions that themselves are already differentiable, like matrix multiplication, transformations, and scalar operations. For the cube and sphere setup, PyTorch3D's built in renderer is sufficient, but for the reflective material or Rubik's cube setups, we must write a large portion of the rendering ourselves with these differentiable functions.

3 Methodology

3.1 Training on a Cube or Sphere

This setup involves having a single object in the scene. In our case, this was a cube or a sphere, but our code generalizes to any object. We simply wish to differentiably render the object mesh with a texture applied.

The cube and sphere setups are identical in terms of their rendering and training pipelines, and are the most straightforward setups we worked on. We heavily utilized PyTorch3D, which provides some basic differentiable rendering and 3D model manipulation capabilities. These built-in functions were enough for these two cases.

PyTorch3D provides objects for cameras, lights, meshes, and renderers, along with the standard PyTorch ML functionality. Our framework for this case starts by setting up each of the above for the desired view, informed by input parameters like camera angle, distance, resolution, etc. We then provide a simple training loop that can be replaced with diffusion. Finally, we define functions to generate figures and videos to visualize the result of training.

3.2 Training on a Reflective Surface

The reflective surface setup includes two objects. The first is a reflective object, which in our case is a cylinder or curved plane, but our code generalizes to any object. The second object is a plane, to which the texture to be optimized is applied. This texture is reflected onto the reflective object to produce the reflected image. As an example, imagine placing a soda can on a photo – the image from the photo gets reflected by the soda can and distorted. We want that distorted image to also take on some recognizable appearance.

The reflective surface setup was a greater challenge than the cube/sphere case, and required modifying some of the existing PyTorch3D rendering source code to fit our needs. The PyTorch3D cameras and lights could be used in the same way as the previous setup; however, the notable difference is that one of the meshes we create will not have a texture, but rather will reflect the scene around it. We can simply give the reflective model a dummy texture and leave the mesh setup the same, but we still need to rewrite the rendering pipeline substantially.

We created a custom reflective renderer which computes the color of each rendered pixel on the reflective object based on the coordinate of the plane texture that should be reflected to that pixel. Given the vector out of the camera onto the reflective surface, we want to find the vector that would be reflected off of the surface and determine if and where it intersects with the flat textured surface. PyTorch3D provides the normal vector at a point on a surface n. We can use this, along with the camera view vector c, in the equation

$$r = c - \frac{2c \cdot n}{\|n\|^2} n$$

where r is the reflected vector. Then, we can use algebra to solve for where the line from the point on the surface in the direction of r intersects the plane $z = 0$.

Here is sample code from the differentiable renderer, using differentiable functions and operators like matrix multiply and grid sampling.

```
N, H, W, K, _ = pixel_normals.shape
viewvec = pixel_coords_in_camera - cameras.get_camera_center().view(N, 1, 1, 1, 3).expand(N, H, W, K, 3)
reflvec = viewvec - 2.0 * pixel_normals * torch.sum(pixel_normals * viewvec, -1, keepdim=True)
reflvec = reflvec / torch.sum(reflvec**2, -1, keepdim=True)**0.5

k = (-pixel_coords_in_camera / reflvec)[..., 1].view(N, H, W, K, 1).expand(N, H, W, K, 3)
planepoint = pixel_coords_in_camera + k * reflvec
planepoint = torch.stack((planepoint[..., 0], planepoint[..., 2]), dim=-1)
planepoint = torch.div(planepoint, plane_scale)

t_H, t_W = texture.shape[1:3]
tex_map = texture.expand(N * K, t_H, t_W, 3)
tex_map = tex_map.permute(0, 3, 1, 2)
pix_coords = planepoint.permute(0, 3, 1, 2, 4).reshape(N * K, H, W, 2)

result = F.grid_sample(tex_map, pix_coords, padding_mode="zeros", align_corners=False)
result = result.reshape(N, K, 3, H, W).permute(0, 3, 4, 1, 2)
result[k[..., 1] < 0] = 0
```

3.3 Training on a Rubik's Cube

The Rubik's Cube setup is similar to the cube setup, where instead of only observing the object from different angles, we also shuffle portions of each face like a Rubik's Cube. Imagine pasting an image on each 3x3 face of a Rubik's Cube and shuffling it. We wish for each of those shuffled 3x3 faces to also show a recognizable image.

While very different from the reflective setup, this also requires writing custom rendering code. In this case, the transformation isn't something continuous like reflection; rather, it is the shuffling of the individual tiles. When shuffling a Rubik's Cube, one cannot simply shuffle the tiles in any way they desire. Each "move" or rotation permutes the tiles in a very specific way.

We needed to hardcode the result of each of these moves, and additionally do this in such a way that the transformation could be composed and performed multiple times to simulate making multiple rotations. All of this needed to be done using differentiable PyTorch3D functions.

Here is a code sample from the `rotate_face` function. This uses differentiable PyTorch functions like transpose and flip, along with reassignment (which is differentiable in PyTorch), to shuffle the tiles (here called `patches`) based on a rotation configuration.

```
def rotate_face(face, patches, cw=True):
    f = rotation_faces[face]
    old_patch = patches.clone()
```

```
edge, n = f["edge"], len(f["edge"])
s = 3 if cw else -3
for i in range(len(edge)):
    patch = patches[0, edge[i][0], edge[i][1]]
    patch = old_patch[0, edge[(i + s) % n][0], edge[(i + s) % n][1]]
    dim = int(not cw) if edge[i][2] == -1 else int(cw)
    if edge[i][2] != 0:
        patch = torch.flip(torch.transpose(patch, 0, 1), (dim,))
    if edge[i][2] == 2:
        patch = torch.flip(torch.transpose(patch, 0, 1), (dim,))
    patches[0, edge[i][0], edge[i][1]] = patch

side, n = f["side"], len(f["side"])
s = -2 if cw else 2
for i in range(len(side)):
    patch = patches[0, side[i][0], side[i][1]]
    patch = old_patch[0, side[(i + s) % n][0], side[(i + s) % n][1]]
    dim = int(not cw) if side[i][2] == -1 else int(cw)
    patch = torch.flip(torch.transpose(patch, 0, 1), (dim,))
    patches[0, side[i][0], side[i][1]] = patch

center = f["center"]
dim = int(not cw) if center[2] == -1 else int(cw)
patches[0, center[0], center[1]] = \
        torch.flip(torch.transpose(patches[0, center[0], center[1]], 0, 1), (dim,))
```

4 Other Contributions

4.1 Generating Results

I also made contributions to generating results. This involved writing new sets of prompts for each setup that would create interesting results, running inference with those prompts, and qualitatively evaluating the results. The most interesting of these steps is writing prompts. One might think any set of prompts would be good enough, but there are several considerations one must take into account to create the best results.

One of the most important is ensuring coherence of the output texture. We need the shapes and textures of the desired views to be compatible. For example, it would be a bad idea to prompt "bubbles" on an overlapping view with "a city skyline", as one consists of purely circles and the other consists of straight lines and angles. There is virtually no way for the diffusion to incorporate aspects of one into the other in a meaningful way, leading to a very unappealing or uninteresting result. However, something like "a garden" and "a bird" can have a lot of overlap in terms of organic shapes and natural colors.

Another consideration is how interesting the set of prompts are as a whole. We could just pick individual prompts that look good and mix and match them, but results are more compelling if they fit a theme or tell a story. "snowy mountain" and "polar bear" follow a consistent theme, while prompts like "museum" with "earthquake" or "caterpillar" with "butterfly" tell a story. One could even use more evocative prompts, like "Joe Biden" and "Donald Trump". Another fun set of prompts we tried was "a crowd of people" and "Waldo" in the reflective cylinder setup. (Figure 2) This created a sort of *Where's Waldo* game, where the players would need to find where on the image to place the cylinder in order to see the reflected image of Waldo.

4.2 Creating User Study

To measure the quality of our results, quantitative evaluations aren't quite enough, as the purpose is to create results that are interesting to humans. To tackle this, I designed a user study and a few metrics for evaluation from the results. This user study would compare different methods of generation, including our own method, previous 2D works that we could apply to the 3D case, and ablations of our method.

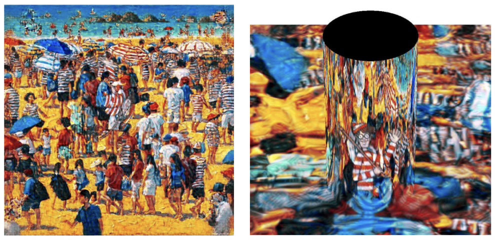
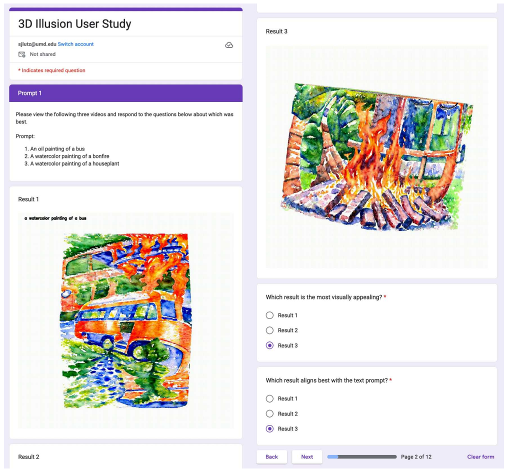


Figure 2: An interesting *Where's Waldo* illusion.

I needed to create a study that would compare the results of these methods and present them to the respondent in a digestible manner. After comparing the capabilities and prices of different options, I opted to create the user study in Google Forms. I needed to evaluate a large set of images and combinations of methods, while ensuring the respondent isn't overburdened with an excessively long survey. The ideal solution would be to randomize the questions that are shown to each respondent, allowing us to get an even set of responses for each result comparison. However, free services like Google Forms don't provide this functionality. Instead, I simply created three different Google Forms, each one with 11 different sets of results and comparisons to evaluate. Then, we could provide respondents with any of the three links, or use a randomized forwarding link from a site like Nimble Links to direct respondents to a random form.

The survey structure was another consideration. I needed respondents to see each of the results from different methods on the same prompt and determine which was their favorite. I chose to display three different methods for each question, and for each one ask "Which result is the most visually appealing?" and "Which result aligns best with the text prompt?".

One hurdle is that our results are videos which transition between the various views of each object; however, Google Forms doesn't support displaying videos without uploading them to YouTube, and even then, the user must play them manually, which creates a more tiresome experience for the respondent. It wouldn't be optimal to only show still images of each view of each result, as this wouldn't properly provide the effect of the images being different views of the same scene. My solution to this was to convert all of the videos to GIFs, which would play automatically and allow users to see all views and the transitions between them.



3D Illusion User Study

sjlutz@umd.edu Switch account

Not shared

* Indicates required question

Prompt 1

Please view the following three videos and respond to the questions below about which was best.

Prompt:

1. An oil painting of a bus
2. A watercolor painting of a bonfire
3. A watercolor painting of a houseplant

Result 1

a watercolor painting of a bus

Result 2

Result 3

Which result is the most visually appealing? *

Result 1 / Result 2 / [✓ Result 3]

Which result aligns best with the text prompt? *

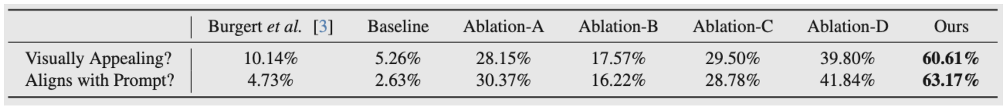
Result 1 / Result 2 / [✓ Result 3]

Figure 3: A sample of the user study.

4.3 Creating Evaluation Metrics

We distributed the user study to friends and colleagues and collected results. Once I had statistics on which method was preferred in each individual case, I needed to compile these results into quantitative evaluations of each method. I devised two different metrics for evaluation – an overall preference score and a pairwise preference score.

The overall metric was: What percent of instances in which a method's result was shown to a respondent was that result chosen? This was computed by dividing the number of times that method was selected overall by the sum of the number of responses for all questions in which it was an option.



	Burgert *et al.* [3]	Baseline	Ablation-A	Ablation-B	Ablation-C	Ablation-D	Ours
Visually Appealing?	10.14%	5.26%	28.15%	17.57%	29.50%	39.80%	**60.61%**
Aligns with Prompt?	4.73%	2.63%	30.37%	16.22%	28.78%	41.84%	**63.17%**

Figure 4: The overall metric results in the paper.

The pairwise metric was: In questions where both methods A and B were displayed, what percent of respondents that picked either A or B picked A? If respondent's selected method C, then we can't conclude anything about their relative preference of A over B, so this case was ignored. We used this metric to evaluate how our method compares with individual alternatives.

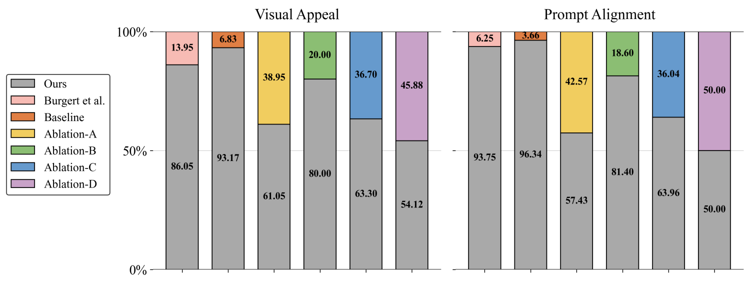


Figure 5: The pairwise metric results in the paper.

4.4 Creating Supplementary Website

As our results are meant to be viewed as videos, the still images in our PDF paper were not as compelling as they could be. This meant that our supplementary website was of crucial importance; it is the only source reviewers have to view the illusions as they should be. I was tasked with creating this supplementary website. I modeled the design off of Visual Anagrams' [3] supplementary website, as they have a similar use case of displaying videos.

The raw videos generated by our diffusion were too large, many on the scale of 100MB or more. When loading these on the website as is, most browsers would stall or simply be unable to load the videos. To fix this, I compressed all of the videos down to around 2-5MB, maintaining essentially the same quality while ensuring the website loads much more quickly.

5 Conclusion

To contribute to this project, I developed differentiable rendering techniques for various scene setups which were crucial for generating results and testing different diffusion methods. These rendering techniques build upon the existing PyTorch3D library and implement functionality for first order reflection and texture permutation. I additionally contributed to the process of result generation, creating a user study, developing evaluation metrics, and building the supplementary website.

6 References

References

[1] Ryan Burgert et al. *Diffusion Illusions: Hiding Images in Plain Sight*. 2023. arXiv: 2312.03817 [cs.CV]. URL: https://arxiv.org/abs/2312.03817.

[2] Yue Feng et al. *Illusion3D: 3D Multiview Illusion with 2D Diffusion Priors*. 2024. arXiv: 2412.09625 [cs.CV]. URL: https://arxiv.org/abs/2412.09625.

[3] Daniel Geng, Inbum Park, and Andrew Owens. *Visual Anagrams: Generating Multi-View Optical Illusions with Diffusion Models*. 2024. arXiv: 2311.17919 [cs.CV]. URL: https://arxiv.org/abs/2311.17919.

[4] Kai-Wen Hsiao, Jia-Bin Huang, and Hung-Kuo Chu. "Multi-view Wire Art". In: *ACM Trans. Graph.* 37.6 (2018). (Proceedings of SIGGRAPH Asia 2018), 242:1–242:11. DOI: 10.1145/3272127.3275070.

[5] Niloy J. Mitra and Mark Pauly. "Shadow art". In: *ACM Trans. Graph.* 28.5 (Dec. 2009), 1–7. ISSN: 0730-0301. DOI: 10.1145/1618452.1618502. URL: https://doi.org/10.1145/1618452.1618502.

[6] Nikhila Ravi et al. "Accelerating 3D Deep Learning with PyTorch3D". In: *CoRR* abs/2007.08501 (2020). arXiv: 2007.08501. URL: https://arxiv.org/abs/2007.08501.