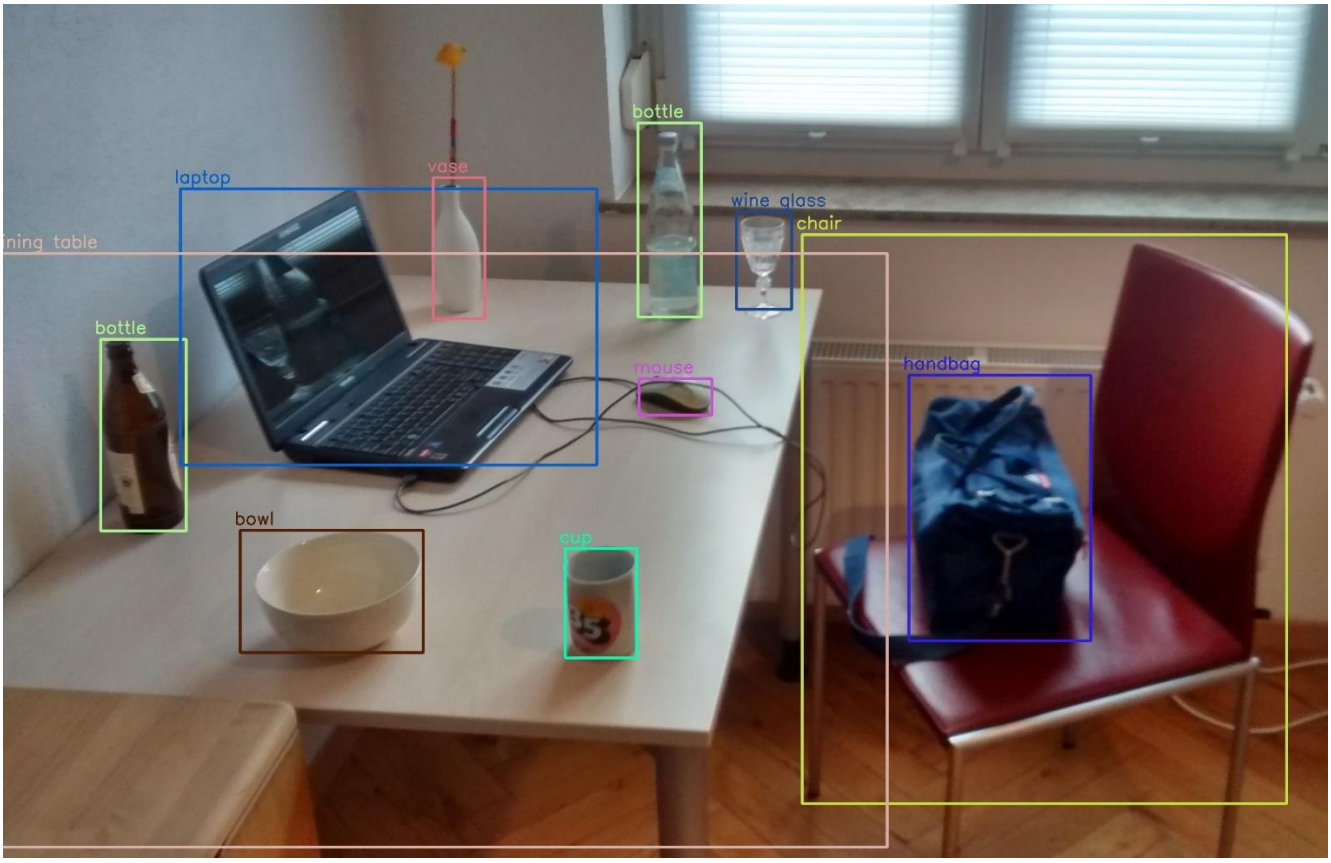


3D SCANNING AS A MEANS OF RAPID TRAINING FOR COMPUTER VISION

Andra Williams

3D Scanning as A Means of Rapid Training for Computer Vision

Andra Williams

Department of Information Science and Technology, University of Nebraska at Omaha

CIST 3000: Advanced Composition for IS&T

Ms. Amanda Gutierrez

27 November 2022

Table of Contents

List of Figures

Executive Summary

Computer Vision is an application of Artificial intelligence that takes in visual data and interprets it in a way that computers can understand. Many intricate steps are involved in training an instance of a computer vision AI, also known as a computer vision model. The four significant steps of training are: obtaining data, labeling data, recognition, and correction. The final two steps, recognition and correction, are frequently explored; however, the first two, obtaining data and labeling data, often take the longest to complete. A potential optimization to these steps is to generate training data from a 3D model.

By combining two applications of artificial intelligence - computer vision and 3D scanning - one can quickly generate training data that is automatically labeled. A 3D scan takes a real-world object and converts it into a 3D model. This 3D model can generate 2D images from any angle relative to the model. A computer vision model can recognize objects more accurately with the wider variations created from the pictures developed from the 3D scan than input data collected manually.

There was more difficulty in generating the training data than expected, but it is proven possible using PLY 3D models. PLY models use points and polygonal faces to build the 3D models, making the 2D projection math much more straightforward than calculating curves. Directly projecting every point and every face takes a lot of resources to complete, so the projection needs to determine what faces are closest to the virtual “camera.” With this setback, it became an intangible goal to run a test comparing a computer vision model trained traditionally against a model taught using 3D scanning during one semester. Hopefully, this paper is proof that it is possible and a start for other researchers to continue the data analysis.

Background Information

Author Knowledge

As a student, there are many opportunities available for self-exploration. Some of Andra Williams' explorations led to research into topics like 3D modeling, 3D scanning, artificial intelligence, autonomous vehicles, and many other varying technological advancements. When presented with a project-based class at Omaha North High School in the 2020-2021 school year, he used these self-explorations to work on a computer vision project that focused on protecting bicyclists riding on the road. During the project, Andra took a more in-depth look into computer vision and computer vision training. One of the issues during that project was limited detection capabilities at varying angles. The final report for the project led to multiple solutions to this problem. One of the proposed solutions was using 3D models to improve the training data size without requiring extensive labor hours for collection. The expectation is that by using a 3D model, one can generate images from various angles and distances, providing a lot more training data for the computer vision model than one could feasibly create manually. This project is a further exploration into that solution.

Intended Audience

The primary audience for this report is innovative researchers, whether for educational or commercial use. There is a low likelihood that curious students looking for self-exploration may find this report or the direct impacts of this research. The report explores the possibility that a computer vision model trained using 3D scans will be more accurate than a model trained on manually collected images. However, there may be a need for more thorough research by the primary audience. An innovative company can decrease the cost of producing AI models through this research which will incentivize further exploration. Since this article intends to be read by academics and corporations, they will already have general background knowledge of the technologies used. Any technical breakdowns included in the report are to be more accessible to the secondary audience. The knowledge that the audience still needs is comprehensive comparison data as proof of effectiveness.

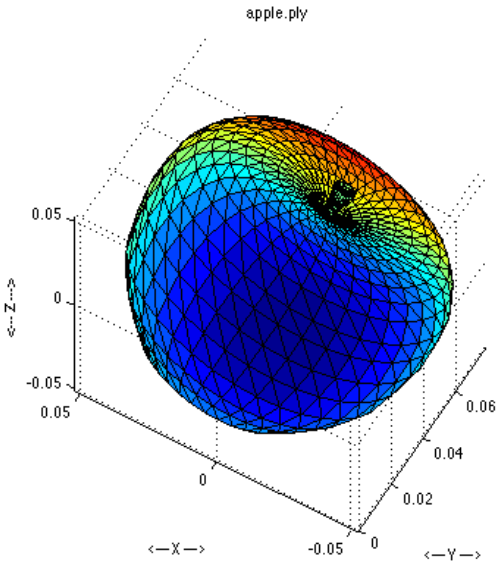
Important Definitions

Before further exploring computer vision and the validity of 3D scanning as a means of procuring training data, a few terms need to be defined. The most fundamental definition for this research is *artificial intelligence* [2], often abbreviated to AI. Depending on who is asked, there are many definitions of AI. However, the concept of artificial intelligence boils down to a computer program programmed to mimic human thought. *Computer vision* [5], or CV, is a specialization of AI that focuses on mimicking human sight. Included in computer vision is *image processing* [7], a process that takes visual data from pictures or video so that a program can obtain meaningful information. An example of image processing is *object recognition* [8], where CV breaks down an image to determine what objects are in it. Object recognition is beneficial for autonomous vehicles, quality control, and even reading QR codes like the one used later in this paper.

Another definition necessary for this research is *3-Dimensional (3D) Scanning* [1]. 3D scanning is a technology that takes a physical object that can be as small as a dime or as large as the Grand Canyon and saves it as data that a computer can process. There are many different types of 3D scanning techniques, like laser triangulation, time-of-flight scanning, and multi-view photogrammetry, to name a few (Arbutina M. et al., 2017). These techniques are explored further in "Types of Scanning Technologies."

Outside of the fundamental definitions, there are a few other essential definitions. An *autonomous vehicle* [4] is a vehicle that moves around while controlled by a computer. An autonomous vehicle can be various types of vehicles like cars, trucks, planes, boats, and UAVs (Unmanned Aerial Vehicles, also known as drones); however, autonomous vehicle most commonly refers to cars for consumer use. The factor distinguishing between autonomous and non-autonomous vehicles is that the computer that controls the car makes its own decisions using artificial intelligence. A *PLY file* [9], as shown in Image 1, is a specific file type for 3D models whose file is formatted in a way that the model I defined by points and polygonal faces. Often, PLY files constrain all of the faces to triangles or, in the case of files in a *voxel* [10] style, squares.

Image 1



Visual representation of PLY file of an apple.
(FSU, 2012)

Computer Vision

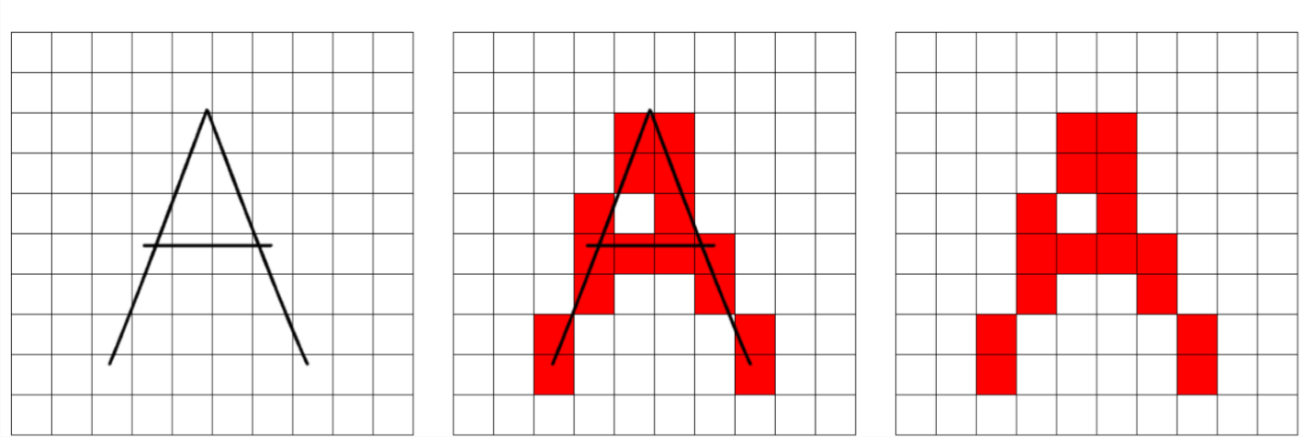
Detailed Definition of CV

IBM's article titled What is Computer Vision states that:

> "Computer vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos and other visual inputs — and take actions or make recommendations based on that information. If AI enables computers to think, computer vision enables them to see, observe and understand" (IBM, What is Computer Vision?).

The definition means that computer vision takes visual data from images and video and processes it to get information like what is in the picture, where things are, how objects are moving, or even obtaining emotion from facial expressions. When people see something, they break what they see into smaller bits of information, which get reconstructed for what is needed (Zhang, 2010). The smaller chunks are shadows, highlights, lines, curves, overall shape, relative size, and many other features. Computer vision is an effort to mimic this as much as possible, but that isn't simple when the cameras used for collecting the visual data save it as a grid of pixels with position and color data.

Images 2, 3, and 4



Original Computer Scan Resulting data

Demonstration of a computer scanning a simple image into a 10 x 10 grid

Images 2, 3, and 4 show the progression of human vision to computer vision. Image 2, the original, shows the letter A as the lines a human would recognize, while image 4, the resulting data, shows the individual pixels a computer would read. It is much harder to read image 4 as the letter A, but knowing the context helps the human brain recognize it. The goal of computer vision is to understand the context, but instead of being able to look at the image as a whole, like the human brain, the picture is searched pixel by pixel. With a more detailed scan, the computer can get more context, but it will always have to predict the lines. Finding the context becomes harder for computer vision models as there are more colors, overlapping lines, shades, and conflicting information. To help simplify these tasks, CV models are more specialized when they need to get more detailed information. The image processing capabilities of Google photos in the

android operating system run multiple separate models for different tasks. There is a model that recognizes text, a model that identifies landmarks, a model that specializes in finding items sold online, and a model that labels plants. This specialization is where training each model separately is essential.

Training a CV Model

Training a computer vision model is the process of teaching the model what information it needs to extract from the visual data. For example, when training a model for object recognition, one would give it tons of pictures of an object and tell the model that all of those pictures are of the same object and during the training process, the model will learn what makes features there are that distinguish it from other objects. Good training data for an object recognition model would include objects that are similar to the intended object. Think of what distinguishes a cup from a mug. Cups can be any size but are usually glass plastic or paper and in the shape of a tapered cylinder, whereas mugs are usually shorter, ceramic, and less tapered or even rounded. Another distinguishing feature is that mugs often have handles. Cups and mugs are very similar but with enough training data a computer vision model can recognize the nuanced differences and maybe pick up on the feature of the handles.

TELUS International wrote an article focused on creating good training data and they laid out 4 steps that are to be repeated until the model is to a satisfactory accuracy. The first step is collecting the data. There are many publicly available datasets like ImageNet and Google's Open Images but "for more practical purposes, collecting proprietary training data, similar to the data required for the final model to run efficiently is probably best" (TELUS International, 2022). This means that if one wanted their CV model to recognize plants, they would need to gather pictures of plants and the more pictures there are, the better. According to ImageNet, the ImageNet dataset contains over 14 million images (2021) and that is just one possible dataset.

The second step is to label the data. The purpose of labeling the input data is to categorize the data in a way that is meaningful to the CV model. For a CV model intended to classify dishes it wouldn't be useful to tell it what a cat is, just that the cat isn't a dish, but it would be useful to tell it whether a knife is a butter knife or a steak knife. This can be a tedious task that can take longer with a larger dataset. It is suggested that the list of labels is small, descriptive, and unambiguous (TELUS International, 2022). If two labels are too similar it may be better to combine them into one broader label.

The third step is to obtain processing power. Since computer vision is mimicking human vision by recognizing individual parts of an image and reconstructing them to match something that it recognies, it takes a lot of processing power to analyze all of the data. Large companies have a lot of available processing power so they can compute everything very quickly, but it still takes a lot of time when they increase the complexity of the models. On the contrary, individuals have less processing power, so it takes a lot longer to process the data. For example, when developing the code for this research project, one of the smaller tasks of rendering images from 3D models would take only 37 seconds to get 1000 images of a simple tetrahedron on a personal computer with only 4 faces and 4 vertices, while a train model, with 121730 vertices and 97384, took hours to get just one image. For each application for CV, there will be a different mix of processing power and processing time.

The fourth step covers multiple different algorithms. There is training, testing, and teaching. During the training algorithm, the CV model goes through all of the input data and tries to learn what the differences are with all of the labels. The testing algorithm uses the CV model

to make predictions on what an image should be labeled as. Using the predicted label, the model is then taught what the correct label is. By going through these three algorithms over and over again, the CV model becomes optimized and more accurate in its predictions. A developer can then repeat this until there is an acceptable margin of error.

All of these processes take a lot of time. There is a need for a lot of clean input data to train with, then the input data needs to be labeled and the larger the dataset, the longer it takes, the processing takes a lot of time, and with how many iterations that need to be done for an acceptable margin of error. If even one of these time-consuming steps can be optimized, new CV models could be developed quicker.

Uses of Computer Vision

Computer vision is a very versatile tool when making anything that interacts with the physical world. IBM reported using IBM Watson to create a highlight reel for the 2018 Masters golf tournament (IBM, What is Computer Vision?). In the same article, IBM also lists autonomous vehicles, and quality control. There are so many applications just in Google's image searching. It can scan images for text, including written text, it can detect notable monuments, it can determine plant species or animal breeds, it can even search an image for purchasable items. Tom Scott, a British YouTuber that focuses on technology and technical knowledge, made a video in August of 2022 that looked at how the United States Postal Service (USPS) uses computer vision to scan the addresses on letters to automate the shipping. In the video it explains that out of all the letters that are sent through the USPS, only 1.2 billion pictures taken of letters went through the final Remote Encoding Center (REC), which is located in Utah. The benefit of using the REC is that every single one of the manually typed addresses is then able to train the computer vision model even further. Ryan Bullock, the manager of Remote Encoding Operations for USPS, said "Back in 1997, when we had 55 RECs open, all of those RECs combined keyed 19 billion images. … Right now, we have about 810 employees here" (Tom Scott). Just by implementing computer vision into the workflow and using manual editing to continue to train the model, the USPS was able to reduce their workforce by a lot, and that is just one application of the technology. Computer Vision is also great at recognizing distinguishing features in a persons face or in their fingerprint. *Facial recognition* [6] software is so widely available that most modern phones now allow facial recognition as a security feature. Apple phones are also able to use this facial recognition to map emojis onto someone's face that reacts to their facial expressions.

3D Scanning

Detailed Definition of 3D Scanning

"3D scanning is a process of analyzing an object from the real world, to collect all the data in order to recreate its shape and appearance, digitally" (Sculpteo, 2022). When breaking down this definition, there are two points of interest: real-world object, and digital recreation. The real-world object could be small like a chair, a mug, or a bug or the real-world object could be large like a building, a city, or a mountain. Depending on the specific technology that is used different levels of precision and different sizes can be achieved. For example, structured light scanning is useful for large objects, but the precision is limited by the spacing of the projected pattern whereas laser scanning is high precision but only useful for much smaller objects because the laser has to slowly scan the entire surface. This balance of scale and precision is primarily what determines which technology is used for each application.

Types of Scanning Technologies

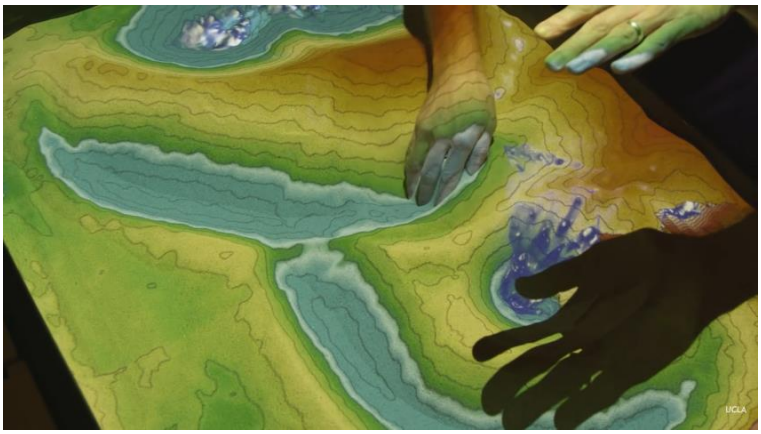
Laser Based Scanning

As the name suggests, laser-based scanning uses lasers to detect a surface to generate a digital recreation of that surface or object. The way this happens is a laser beam is sent out from the scanning device, the laser hits the object and is reflected back at the device, the laser is then detected with a sensor, and finally using the angle that the laser is returned to the sensor and some trigonometry, the distance between the scanner and the surface is calculated. Since the lasers are hyper focused there is very little for the light to spread. An unfocused light would follow the Inverse Square Law where the gets less intense the further it moves. One such instance of using lasers at a greater distance is using satellites to scan the surface of the Earth.

Structured Light Scanning

Structured light scanning uses a projector to project a pattern, usually a grid of dots or stripes, onto a surface and a corresponding camera uses the distortion of the pattern to determine the shape of the surface (Arbutina M. et al., 2017). Because the pattern is known, the depth of the surface can be calculated using trigonometry, similar to how laser-based scanning is done. The major difference between laser based scanning and structured light scanning is that the laser covers each point individually while the structured light scanning covers an entire surface at once. Scanning the entire surface leads to an increase in speed, but because the projector and camera/cameras are generally very bulky, it is much easier to scan a flatter surface. One use of this technology is a projection sand table or *augmented reality sandbox* []. An Augmented Reality Sandbox (UCLA) is a little unique in that the pattern changes in real-time to reflect the table's surface like it is shown with images 5 and 6. The sandbox is great at showing geological features because the sand provides a physical representation that can be manipulated, and the sand reacts based on the other shapes that ere in the sand. To create a mountain in the sand, there must also be a valley. In the UCLA sandbox there is a feature that demonstrates how water will react to those features. The camera recognizes certain hand gestures using computer vision and using the 3D scan of the surface of the sand the projector shows water from where the hand gesture is recognized, and it changes based on how the water would run off of the peaks and valleys.

Images 5 and 6



Screen capture and QR code for a demonstration of UCLA's Augmented Reality Sandbox https://youtu.be/CE1B7tdGCw0 (UCLA, 2015)

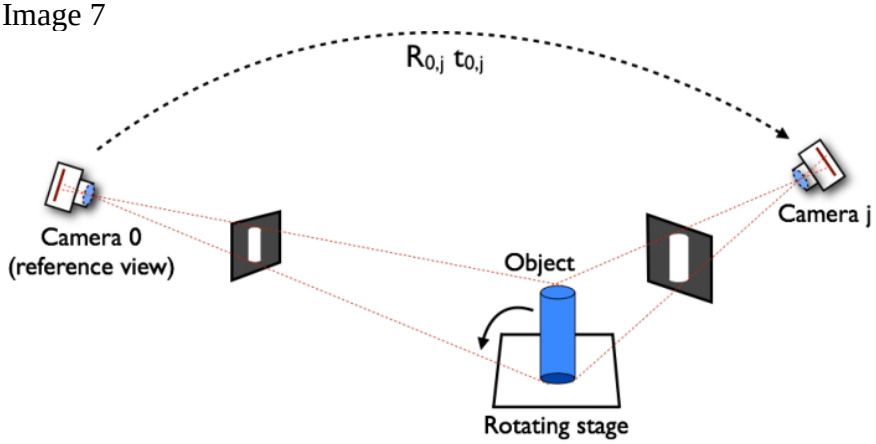
Time-of-Flight Scanning

Another method that is similar to laser-based scanning is Time-of-Flight scanning, also known as ToF scanning. These scanners use the speed of light to calculate how far an object is from a laser and sensor pair (Arbutina M. et al., 2017). Sound-based distance sensors work in a very similar way but ToF scanning is a lot more accurate since the laser is focused instead of being susceptible to spreading out. The reduced spread and increased speed allow the accuracy of the distance calculation to be increased while still allowing the surface or object to be scanned from much further.

Multi-View Photogrammetry Scanning

Multi-view photogrammetry is a process that uses multiple 2-dimensional pictures to generate a 3-dimensional model. The model is created by recognizing how the images are skewed from each other. Multi-view photogrammetry is the simplest to set up, but it requires a lot more processing than other methods. Models that are generated from this method are also very dependent on the clarity of the pictures that are inputted.

Image 7



Demonstration of photogrammetry scanning a cylindrical object. (Martinello, 2016)

Research Summary

Intended Steps

When researching began there were certain expectations of what could have been done, but unfortunately the preparations were taking longer than expected. The original plan was to spend at most 2 weeks to develop the code that generates the projections to train the computer vision model from. Once the 3D to 2D conversion was established, the plan was to use the multi-view photogrammetry to take pictures of common household objects like chairs, cups, tables, and trashcans. Using these generated models, the program can generate the training images. Once the training images are generated, they can be used to train a computer vision model using OpenCV. Using the Holland Computing Center, that is available through the University of Nebraska Lincoln, multiple CV models could be generated. Some of the CV models would be trained using the original images for the multi-view photogrammetry, while others would be trained using the images that were generated from the models. The two groups of models could be compared to see how accurate they are. The comparison of their accuracy could be used to show how effective using 3D scans to train computer vision could be.

Completed Steps

3D Scanning Method

The scanning method that had been chosen for this research is multi-view photogrammetry. This was chosen because it is inexpensive and easy to do on a cell phone. Another benefit of this method is that many services that complete the model generation can output as a PLY file. Multi-view photogrammetry is also capable of scaling to various sizes depending on what is needed. The scans could cover large areas while still being able to scan smaller objects.

Developing Images

There was a lot of work that was put into developing the code that would produce images from a 3D model. Using the basis from a YouTube video that projects points onto a 2D plane, the rotation of the 3D model was calculated which led to images of simple shapes like cubes and tetrahedrons. The projections use trigonometric formulas to convert from 3D points to 2D points. The comments in the code are written to help explain the function of each portion of the code.

```
#This matrix converts a 3D point to a 2D point.
projection_matrix = np.matrix([
    [1, 0, 0],
    [0, 1, 0],
    [0, 0,0]
])

#The following three functions take a rotation angle and adjusts the point's
X, Y, and Z based on the rotations in each angle
def rotation_x (angle_x):
    return np.matrix([
         [1, 0, 0],
        [0, math.cos(angle_x), -math.sin(angle_x)],
        [0, math.sin(angle_x), math.cos(angle_x)]])

def rotation_y(angle_y):
```

```
    return np.matrix([
         [math.cos(angle_y), 0, math.sin(angle_y)],
        [0, 1, 0],
        [-math.sin(angle_y), 0, math.cos(angle_y)]])

def rotation_z(angle_z):
    return np.matrix([
         [math.cos(angle_z), -math.sin(angle_z), 0],
        [math.sin(angle_z), math.cos(angle_z), 0],
        [0, 0, 1]])

#It is important to use math.cos() and math.sin() instead of the built in
cos() and sin() because the built in functions return complex numbers which
make matrix multiplication difficult.
```

While the above code handles converting 3D points into 2D points, the following code takes any list of points that represents a 3D model and takes a given number of pictures at various angles while filtering blank images.

```
#This function takes a list of 3D points as array [x, y, z], a list of faces
that is defined by two lists (a list of points stored as indexes in the
points array and a color stored as RGB values), a number denoting how many
images are generated, and a name to label the images.
def render_faces(points, faces, folder_size, name):
    pygame.display.set_caption("Projection of " + name)

    folder_name = name +"_Projection"
    parent_dir = os.getcwd()
    #This part of the code is used to make sure that multiple projections can
be run without overwriting preexisting files
    found = True
    count = 0
    while found:
        try:
            path = folder_name + str(count)
            os.mkdir(os.path.join(parent_dir, path))
            found = False
        except:
            count += 1

    #These variables set the default angle and translation for the images;
they are changed by a set amount each time an image is drawn.
    angle_x, angle_y, angle_z = 0, 0, 0
    translate_x, translate_y, translate_z = 0, 0, 0
    #The switch variables go back and forth between 1 and -1 to ensure that
the object stays within the screen.
    tSwitch_x, tSwitch_y, tSwitch_z = 1, 1, 1

    #These variables set a limit for how much the object is translated. The
intention is that there will be an equation to constrain the object to the
screen.
    tMax_x, tMax_y, tMax_z = 5, 5, 5
```

```
    #This loop finds the largest widest direction of the object to set the
scale of the object to allow the largest possible variation without the
object leaving the screen.
    max_x, max_y, max_z = -100000000, -100000000, -100000000
    min_x, min_y, min_z = 100000000, 100000000, 100000000
    for point in points:
        if point[0] > max_x: max_x = point[0]
        if point[1] > max_y: max_y = point[1]
        if point[2] > max_z: max_z = point[2]
        if point[0] < min_x: min_x = point[0]
        if point[1] < min_y: min_y = point[1]
        if point[2] < min_z: min_z = point[2]

    x_diff = max_x - min_x
    y_diff = max_y - min_y
    z_diff = max_z - min_z

    xyz_maxDiff = 0
    if x_diff > y_diff and x_diff > z_diff :
        xyz_maxDiff = x_diff
    elif y_diff > x_diff and y_diff > z_diff:
        xyz_maxDiff = y_diff
    elif z_diff > y_diff and x_diff < z_diff:
        xyz_maxDiff = z_diff
    else:
        xyz_maxDiff = x_diff

    sizeMin = 0
    if WIDTH < HEIGHT:
        sizeMin = WIDTH
    else:
        sizeMin = HEIGHT

    scale = (xyz_maxDiff) * (.10 * sizeMin)
    print(scale)

    #The projected_points and translated_points variables are set with an
empty list the same size as the list of points which ensures that there is
space for the calculations to output without changing the original list.
    projected_points = [
        [n,n] for n in range(len(points))
    ]
    translated_points = [
        [n,n,n] for n in range(len(points))
    ]

    clock = pygame.time.Clock()

    #This is the main loop of the function. The loop iterates until the list
of images is the requested size.
    image_count = 0
    while image_count < folder_size:
        clock.tick(30)

        #This allows the screen to be closed early with the escape key.
        for event in pygame.event.get():
```

```
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    pygame.quit()
                    exit()

        #------------------DRAWING---------------
        #The screen gets reset every time there is a new render
        screen.fill(WHITE)

        #The original points are translated using the corresponding
        for p in range(len(points)):
            translated_points[p-1][0] = translate_x + points[p-1][0]
            translated_points[p-1][1] = translate_y + points[p-1][1]
            translated_points[p-1][2] = translate_z + points[p-1][2]

        i = 0
        for point in translated_points:
             #Reshape forces each point to be a vertical array which allows
it to be multiplied by the rotation matrices.
            rotated2d = np.dot(rotation_x(angle_x),
np.matrix(point).reshape(3, 1))
            rotated2d = np.dot(rotation_y(angle_y), rotated2d)
            rotated2d = np.dot(rotation_z(angle_z), rotated2d)
            #The next step is to use the dot product to project x, y, z of 3d
shape to x, y of the screen.
            projected2d = np.dot(projection_matrix, rotated2d)

            x = float(projected2d[0][0] * scale) + circle_pos[0]
            y = float(projected2d[1][0] * scale) + circle_pos[1]

            projected_points[i] = [x,y]
            i += 1

        #The next loop draws each of the faces in the color.
        for face in faces:
                corners = []
                for i in face[0]:
                    corners.append(projected_points[i])
                #This extra append adds the first corner to the end of the
list which is needed for drawing the polygons.
                corners.append(corners[0])
                pygame.draw.polygon(screen,pygame.Color(face[1]),corners)

        #The following allows the calculation of the average which is used to
ensure that the object is within the screen.
        x_sum, y_sum, z_sum = 0, 0, 0
        for p in range(len(translated_points)):
            x_sum += translated_points[p-1][0]
            y_sum += translated_points[p-1][1]
            z_sum += translated_points[p-1][2]

        average_point = [x_sum/len(translated_points),
y_sum/len(translated_points), z_sum/len(translated_points)]
```

```
        rotated_average2d = np.dot(rotation_x(angle_x),
np.matrix(average_point).reshape(3,1))
        rotated_average2d = np.dot(rotation_y(angle_y), rotated_average2d)
        rotated_average2d = np.dot(rotation_z(angle_z), rotated_average2d)
        projected_average2D = np.dot(projection_matrix, rotated_average2d)

        #The variables aX and aY are used for making the average relative to
the center of the screen similar to the individual points.
        aX = float(projected_average2D[0][0] * scale) + circle_pos[0]
        aY = float(projected_average2D[1][0] * scale) + circle_pos[1]

        projected_average = [aX,aY]

        angle_x += .125
        angle_y += .015
        angle_z += .025

        if (translate_x * tSwitch_x) >= (tMax_x * tSwitch_x):
            tSwitch_x = tSwitch_x * -1

        if (translate_y * tSwitch_y) >= (tMax_y * tSwitch_y):
            tSwitch_y = tSwitch_y * -1

        if (translate_z * tSwitch_z) >= (tMax_z * tSwitch_z):
            tSwitch_z = tSwitch_z * -1

        translate_x += random() * .005 * tSwitch_x
        translate_y += random() * .0025 * tSwitch_y
        translate_z += random() * .00125 * tSwitch_z

        image_label = path + "/" + name + "_projection" + str(image_count) +
".jpg"

        #The screen is updated and saved as an image if and only if
projection is on screen. This also ensures that the image counter only
changes when an image is saved.
        if(projected_average[0] > WIDTH *.9 or projected_average[0] < WIDTH *
.1 or projected_average[1] > HEIGHT * .9 or projected_average[1] < HEIGHT *
.1):
            tSwitch_x = tSwitch_x * -1
            tSwitch_y = tSwitch_y * -1
            tSwitch_z = tSwitch_z * -1
        else:
            pygame.display.update()
            pygame.image.save(screen, image_label)
            image_count += 1
```

The code is still incomplete because there is no optimization that ensures that only faces that are closest to the virtual camera are shown. This slows the calculations down a lot because there are a lot of unnecessary faces that are drawn and there is also no guarantee that the faces that are drawn are in the "front of the image." A lot of this code is based off of a video tutorial created by Pythonista_. The tutorial has been modified to work with the formatting of the faces as well as the points. Another modification to the tutorial is the constant translations and ensuring that the object is in the camera.

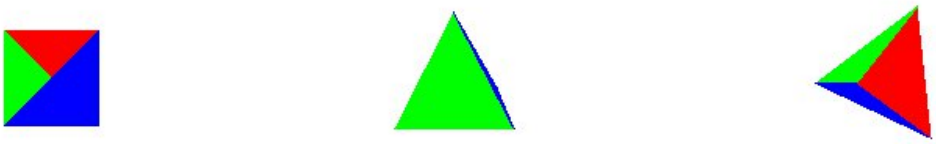
For the function to be used on PLY files, another function is necessary for the extraction of the information from the file.

```
#The function takes in a PLY file and a number that represents how many
images need to be generated
def ply_render_faces(file, length):
    points = []
    faces = []
    #This opens the PLY file and interprets the data using PLYData from the
plyfile library.
    with open(file, 'rb') as f:
        plydata = PlyData.read(f)

    #Some ply files save the color data in the vertices while others save the
color data in the faces. By checking if the 'red' attribute is given to the
vertices, the program can behave accordingly.
    if 'red' in plydata['vertex']:
        for i in range(len(plydata['vertex']['x'])):
            points.append([plydata['vertex'][i][0], plydata['vertex'][i][1],
plydata['vertex'][i][2]])

        #For every face that has a list of vertices, this loop gathers all of
the vertices and finds the average color of the vertices.
        for i in range(len(plydata['face'])):
            corners = []

            red_avg, green_avg, blue_avg, count = 0, 0, 0, 0
            #The first number, index 0, in the face is the number of corners
so count from 1 to the first number minus one.
            for j in range(len(plydata['face'][i][0])):
                corners.append(plydata['face'][i][0][j])
                red_avg += plydata['vertex']['red'][plydata['face'][i][0][j]]
                green_avg +=
plydata['vertex']['green'][plydata['face'][i][0][j]]
                blue_avg +=
plydata['vertex']['blue'][plydata['face'][i][0][j]]
                count += 1

            red_avg = red_avg / count
            green_avg = green_avg / count
            blue_avg = blue_avg / count

            color = (red_avg, green_avg, blue_avg)
            faces.append([corners,color])
            #Each face is now a list of corners followed by an RGB value.

        #The name of the folder is defined by the file name without the file
type. So if the file is "square.ply", the folder will be named "square".
        folder_name = file[0:(len(file)-4)]
        projection.render_faces(points, faces, length, folder_name)

    #The next section of the code replicates the previous section. The only
difference is that there is no need to find the average color of each vertex.
    if 'red' in plydata['face']:
```

```
        for i in range(len(plydata['vertex']['x'])):
            points.append([plydata['vertex'][i][0], plydata['vertex'][i][1],
plydata['vertex'][i][2]])

    for i in range(len(plydata['face'])):
        corners = []
        color = (plydata['face'][i][1], plydata['face'][i][2],
plydata['face'][i][3])

        for j in range(len(plydata['face'][i][0])):
            corners.append(plydata['face'][i][0][j])
        faces.append([corners,color])

    folder_name = file[0:(len(file)-4)]
    projection.render_faces(points, faces, length, folder_name)
```

The above code allows the input of a PLY file to be used to generate the images. Given a PLY file that defines the 3D model, the model is then projected onto the screen and a certain number of pictures are generated in various angles. The optimal code would be able to run in the terminal automatically taking a PLY file and creating a folder of images that could be used as training data for a CV model.

Images 8, 9, and 10



Projections of a tetrahedron that were generated from a PLY file. Generated as images 0, 17, and 47.

Summary of Research

Artificial intelligence is a field of study that encompasses a broad range of topics whose goal is to make computers process real-world data similar to how a person would, but much faster. This covers computer vision which mimics the human brain's ability to process visual data, whether that is still images or video data. When training a computer vision model, there are 4 steps that are outlined by TELUS International: data collection, data labeling, finding processing power, and training/testing/teaching. The final two steps are generally quickly automated because most research or development teams already have direct access to large processing systems and the training, testing, and teaching algorithms are able to run while the research teams are working on other projects or working on collecting more data for the next run of the training loop. Comparatively, the first two steps take a lot of man-hours to complete. Someone needs to either take a lot of pictures or gather pictures from online databases like the ones that are provided with OpenCV, a python library that makes it easier for an individual to develop computer vision at home. When the training data is gathered, someone then needs to sort through them and give them a descriptive label where data with the same label are of the same or similar enough objects. That means that cups may be a label and mugs are another label but that also means that cups and mugs could also be under a single label called drinking containers. This discrepancy comes from personal decisions, which can also make the training data hard to use. If images are automatically generated and labeled, it would take a lot less time to get accurate computer vision models.

By introducing 3D scanning technology, specifically multi-view photogrammetry, a research or development team would be able to take multiple images of the same object, label the one object, and use computing power to generate as much training data as possible. When working on proving the feasibility and effectiveness of this approach, there were a few hiccups. There was a lot more trial and error when programming the 3D to 2D projection algorithms, and the algorithm needed to be optimized to only display the parts of the model that are closest to the screen.

With the limited time frame, the direct comparison of traditionally trained CV models in relation to CV models trained using the 3D scans needed to be scrapped. Instead of doing the comparison, there was a lot of research done on PLY file formatting and 3D projections. These same calculations could also be used as a foundation if humanity ever found a way to create 4D images outside of the simple shapes like the hypercube and the hypersphere (Eckstein).

With anyone that wishes to continue this research, they would need to develop the optimizations that limit what is drawn on the screen and collect meaningful data of how effective the conversion to 3D training would be.

References

Arbutina, M., Dragan, D., Mihic, S., & Anisic, Z. (2017). Review of 3D Body Scanning Systems. Acta Technica Corviniensis - Bulletin of Engineering, 10(1), 58–65.

Best Uses of 3D Scanning and Its Applications. Capture 3D. (n.d.). Retrieved September 11, 2022, from https://www.capture3d.com/knowledge-center/blog/best-uses-of-3d-scanning-software

Eckstein, L. (n.d.). *The Hypersphere. Shapes of Space: The Hypersphere*. Retrieved November 26, 2022, from http://www.math.brown.edu/tbanchof/STG/ma8/papers/leckstein/Cosmo/sphere.html#:~:text=A%20hypersphere%20is%20the%20four,the%20hypersurface%20of%20a%20hypersphere.

FSU. (2012). *Apple* [PNG]. https://people.sc.fsu.edu/~jburkardt/data/ply/ply.html

Martinello, M. (2016). *No Title* [PNG]. Research Gate. https://www.researchgate.net/figure/Example-of-multi-view-system-for-3D-reconstruction-Precise-camera-position-and_fig1_307515923.

MTheiler. (2019). *Detected-with-YOLO--Schreibtisch-mit-Objekten* [JPEG]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Detected-with-YOLO--Schreibtisch-mit-Objekten.jpg

Pythonista_ (19 March 2021). *How to make a 3D projection in Python | Rendering a cube in 2D! (No OpenGL)* [Video]. Youtube. https://www.youtube.com/watch?v=qw0oY6Ld-L0&t=2s

Sculpteo. (2022, July 25). *How Does 3D Scanning Work?* Sculpteo. Retrieved October 23, 2022, from https://www.sculpteo.com/en/3d-learning-hub/basics-of-3d-printing/what-is-3d-scanning/

Stanford Vision Lab, Stanford University, & Princeton University. (2020). ImageNet. Retrieved October 23, 2022, from https://image-net.org/index.php

TELUS International. (2022, March 4). *A Guide to Building Training Data for Computer Vision Models.* TELUS International. Retrieved September 11, 2022, from https://www.telusinternational.com/articles/guide-to-building-training-data-for-computer-vision-models

Tom Scott. (8 August 2022). *How the US Postal Service reads terrible handwriting* [Video]. Youtube. https://www.youtube.com/watch?v=XxCha4Kez9c

What is Computer Vision? IBM. (n.d.). Retrieved September 11, 2022, from https://www.ibm.com/topics/computer-vision

Xu, S., & Zhang, X. (2020, November 30). *Research on Image Processing Technology of Computer Vision Algorithm*. Retrieved September 11, 2022, from https://ieeexplore-ieee-org.leo.lib.unomaha.edu/document/9270498.

Zhang, B. (2010, October 11). *Computer Vision vs. Human Vision*. Retrieved September 11, 2022, from https://ieeexplore-ieee-org.leo.lib.unomaha.edu/document/5599750.

Glossary

3-Dimensional (3D) Scanning
3D scanning is an application of artificial intelligence that takes a real-world object and creates a 3D model that can be visualized and processed with a computer.

Artificial Intelligence
Artificial Intelligence is any program that is designed to mimic the human brain by making decisions based on input information. Advanced artificial intelligence makes better decisions based on prior experience or training.

Augmented Reality Sandbox
An augmented reality sandbox or a topology table is often a demonstration that shows how land masses affect each other and how land masses affect water.

Autonomous Vehicle
Autonomous vehicles are vehicles that use artificial intelligence to make decisions on how to move relative to the physical world. These vehicles have no need for a driver, but the current state of the technology requires a driver to assist in emergency situations.

Computer Vision
Computer vision is a computer program that takes visual data, pictures and videos, and obtains meaningful information from them.

Facial Recognition
Facial recognition is an application of computer vision that recognizes features of the human face to either recognize that there is a face or recognize whose face is shown in the visual data.

Image Processing
Image processing is a process of taking a picture and modifying it so it becomes something else. The topic of image processing covers both photo editing and gathering data from images.

Object Recognition
Object recognition is an application of computer vision that identifies familiar objects within visual data. Facial recognition is a specific type of object recognition.

Ply File
Polygon files or PLY files is a file format that saves 3D data as a list of points and a list of faces referencing those points. Each face is a flat surface that is in the shape of a polygon, usually a triangle.

Voxel
Voxels are often referred to as the 3D equivalent of a pixel. A 3D model is referred to as a voxel model if it can be broken down into equally sized cubes. All of the faces of the model are either cubes or rectangles that have a uniform spacing.