Creating Photo-Realistic Synthetic Scenes for Machine Learning Tasks in Unity

Brianna Butler
Computer Science B.S., Duke University
Intelligent Interactive Internet of Things (I3T) Lab @ Pratt
Durham, NC
bb274@duke.edu

***Abstract*—Using Unity3D's 3D game rendering workspace, we generated photorealistic environments to test the engine's object recognition (and other) machine learning algorithms. Through Unity, we can test the deep neural network based object recognition classifier performance under conditions not possible in the real world: different lighting, object positions, user positions, number of users, multiple angles, obstructions, etc. We will utilize scenes as close as possible to being photorealistic with features such as reflections and transparent objects. Then we developed a pipeline that ran various classifiers on these scenes and assessed their performance. After that, we showed that the game-based Unity engine allowed us to test conditions that we could not test with existing data sets from purely reality-based datasets.**

***Index Terms*—Deep Neural Networks, Object Detection, Augmented Reality, Game Engine**

I. Introduction

Technology is advancing at an exponential rate, and due to this, technologies out of science fiction films of the past are being used on a daily basis by the average citizen. A recent technological advancement that is making big waves is the use of augmented reality in daily life. The ultimate goal of augmented reality is to provide users with an immersive experience of a virtual world blended into the reality they are currently in. Augmented reality heavily relies on accurate mapping of the virtual world in the real world and low latency of these applications.

Examples of the uses of augmented reality range from tasks as fun as catching Pokemon in your room in Pokemon Go, to as educational as training new workers in hands on jobs, to as critical as military combat training.

The military utilizes the BARS, or the Battlefield Augmented Reality System, which can be used to transfer three-dimensional tactical and strategic information from a command center to individual war fighters operating in their own current environment.

BARS is currently being developed to tackle common dangerous issues of combat [18]:

- complex 3D environments and dynamic situations
- loss of line-of-sight contact to team members
- accurate tracking and registration systems

Microsoft is currently developing the HoloLens, an augmented reality headset that can be used to train inexperienced workers in a new job - a virtual trainer is there to guide them through the steps needed to do well at their work. Real-time views with experts in remote locations help users with the hands-free or remote assist versions of HoloLens. The Dynamics 365 features of the HoloLens services allows users to capture their current environment on their phone and integrate it with the Power Platform or other partnering applications.

Fig. 1. Example of the military's BARS backpack.

Fig. 2. Example of Microsoft's HoloLens at Work.

Companies currently partnering with Microsoft to help develop the use of HoloLens and its augmented reality features include CAE, GIGXR, and Hevolus [17].

Because of the mass use of augmented reality, and how

Fig. 3. Image of reality-based testing on an artificial scene.

important it can be in certain dangerous situations, AR (augmented reality) needs to be accurate for users [17].

Commonly used examples of the importance of augmented reality is how augmented reality's object recognition properties can be the difference between life or death in the case of autonomous drones, search- and rescue robots, and self-driving cars. In the case of the death of Elaine Herzberg in 2018. Herzberg was struck and killed by a self-driving Uber SUV as she was riding her bike at night. The driver of the vehicle was unaware of Herzberg because she was on her cellphone. The vehicle had trouble detecting what Herzberg was at the time of night - she was classified as an unknown object, so the vehicle continued driving. Only the driver of the vehicle would have been able to stop the car when the car was unable to classify the object. Had the SUV been able to detect Herzberg in low lighting , the SUV would have known to hit the brakes to avoid a collision. However, due to the failure of the object recognition system, a woman lost her life [7].

In this scenario, the detectors failed to recognize Herzberg due to her being an object out of the normal distribution of training for the detection system. What if we were to use virtual worlds to train these detection systems in ways not possible in the real world?

A big proportion of this study was highly motivated and inspired by [4] and [5]. In their study, they also used self-created scenes to test object recognition algorithms. This will be further discussed in the related works section of the paper.

II. Related Work

Two articles in particular [4], [5] heavily inspired and motivated this study.

In [5], the idea of using synthetic scenes to test deep neural networks is explored. In this paper, instead of YOLOv4 like us, they used Inception v3 DNN from the PyTorch model zoo. The renderer they chose was ModernGL. They constructed their scenes with 3D objects, lighting, background scene and parameters, and estimated the pose changes of their main object (a school bus) that caused the classifier to misclassify by approximating gradients via finite differences, or back propagating through a differentiable render.

In their study, misclassifications uniformly covered the pose space, objects were commonly misclassified as the wrong objects, and correct classifications were highly localized in the rotation and translation landscape. The paper evaluated v3 in many out-of-distribution positions, transformations, and backgrounds and identified possible biases in the datasets to train v3 themselves: there is a likely chance that the collected datasets represent the photographer and the aesthetics of the photographer themselves. However, for their study, the scenes they created were not truly photorealistic, and they did not interact "naturally" (physics-wise) with their backgrounds.

Their overall contributions to the study of DNNs and virtual worlds included: the identification of real world and virtual out-of-distribution poses, the introduction of a new moth for testing computer vision DNNs with 3D renders and 3D models, and they provided insight into the failures of modern object recognition algorithms. Similarly to our study, they found that their DNN struggled to identify the objects in their scene when the objects were rotated in directions that differed from where they would normally be in the real world.

In [4] they developed photorealistic scenes through Project Chrono (a physics engine) and Opposite Renderer2.

In the paper, they discussed a way to produce realistic synthetic scenes through a physic engine (like Unity) and sensible camera trajectory generation. In addition to the scenes,

they made sure to implement artificial motion blur that could appear in reality as well. Motion blur commonly occurs in photographs and with the naked eye. In the paper, they also introduced the idea of applying random/abnormal textures to everyday/household objects.

However, the paper did have a few weaknesses. There is more work needed on curating accurately scaled objects, and ensuring accurate and consistent labels on an object dataset. Automated systems mistakenly found mailboxes from ShapeNets when searching for the object category 'box'. They also only focused on static scenes – dynamic scenes with more "soft bodies" would allow for a more realistic representation of the real world.

They also did not account for the mass or friction coefficient of objects that could affect their interactions with objects in real life.

Overall, their main accomplishments relating to our study were that they accomplished producing realistic synthetic pixel-by-pixel labelled data, and they bridged the gap between reality and artificial scenes to allow manipulation of realistic data sets not possible in reality [4].

III. Implementation

A. Unity

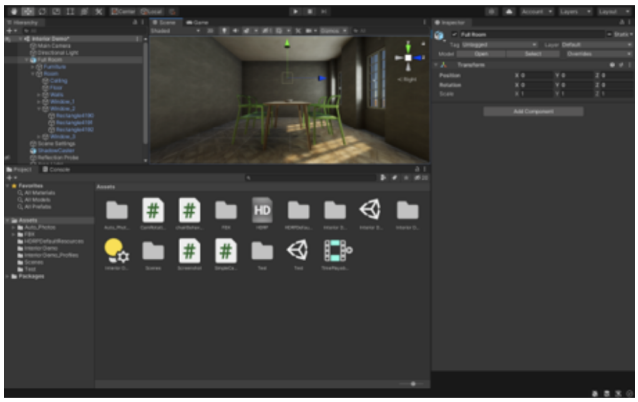

Fig. 4. Screenshot of the whole editor window of Unity.

The majority of the project was developed, ran, and analyzed on Unity, one of the leaders in the industry of game engines for personal uses and businesses alike. Unity gives its users the ability to create 2D and 3D games, and offers scripting options in C (used in our study) for both plugins and games, as well as the option to use drag and drop features. In Unity, you can import and move around/place assets and create or import animations to use within the game.

Unity allowed us to manipulate nearly any feature of the scene we wanted to for the project. In a study that relied so heavily on the availability and creation off abnormal scenes for DNN testing, Unity makes it easy to create scenes with extreme object rotation, lighting, backgrounds etc. Unity allowed us to make any arrangement of the scene that we wanted. [16]

The main features of Unity that will help us the most in our project is:

- Unity's Asset Store: This is where we will obtain our objects and scenes to test YOLOv4's object recognition technology on.
- Editor scripting: We will use the scripts to automate the processes of in-game screenshots, camera rotation, and object rotation.
- Unity's object manipulation options: the ones we focus on in our study is the ability to manipulate scene lighting and shadows, object rotation and positioning in the scene, the creation and destruction of visible objects, and the material and texture editors for objects in Unity

Most Unity applications Unity's scripting features to respond to player input and arrange for events in-game to happen when they should.

B. Scenes

Rather than create our own scene with our own created objects, we decided to use a premade, free scene available on the (site where the scene is from). We went in the direction of a premade scene due to our lack of experience making realistic scenes and objects ourselves - having photo-realistic objects and scenes are key to the project and the object detection training. The first scene we utilized was the Interior Rendering Tutorial from the Unity Asset Store.

The scene consists of a closed room with 2 windows, 3 paintings, 4 chairs, 1 bowl, and one dining table:

- Windows: Generic white windows that show the environment outside of the window and also the source of light for the scene.
- Paintings: Non-specific, unidentifiable paintings.
- Chairs: Green plastic chairs all arranged around a table in the middle of a room.
- Bowl: Gray ceramic-looking bowl placed on the exact center of the table.
- Table: Light-colored wooden table placed in the exact center of the room.
- Walls: The room is enclosed within 4 gray walls.
- Floor: The floor appears to be a gray tiled floor.

Another feature of the room is a door frame in the back right corner showcasing the outside of the room.

In addition to the default objects in the Interior Rendering Tutorial Scene, we decided to test the results of YOLOv4 on a 3D model of a skull placed in the scene. The model was provided by Tim Scargill, ECE Ph.D student from Duke University, and fellow IÎT Lab member, who 3-D printed a skull who he had a 3D model of using Duke's Co-Lab.

C. Deep Neural Network

For the study, we decided to test the accuracy of YOLOv4's object recognition capabilities (YOLO standing for 'You Only Look Once'). YOLOv4 was the choice deep neural network due to its reputation as one of the most precise and fastest recognition systems available as of late.

YOLOv4 runs twice as fast as EfficientDet although their respective performances can be considered comparable. YOLOv4, compared to the previous version, YOLOv3, has

Fig. 5. Image of Interior Rendering Tutorial main scene used for this study

Fig. 6. Image of skull model, which was later placed in the Interior scene.

improved AP and FPS by 10% and 12% respectively. It can predict up to 9000 object classes, predict multiple objects at once, and even draw bounding boxes around the objects it recognizes in a given image.

It can also be trained to detect custom classes and objects. The. detector can be trained and used on a conventional GPU with 8-16 GB-VRAM. YOLOv4 is the second-most up to date version of the famous series of object recognition programs [20].

We chose to use YOLOv4 rather than YOLOv5 (the most up-to-date model in the YOLO series). Although YOLOv5 has a faster training process, YOLOv4's performance cane be optimized to achieve higher frames per second. Another difference between the two is that YOLOv5 is implemented through PyTorch and and YOLOv4 is through Darknet, which is why top-accuracy research is more likely to continue with YOLOv4 rather than YOLOv5.

YOLOv5, overall, is better for real time object detection, but YOLOv4 is more accurate, and our research project revolves around accuracy [21].

D. Screenshot Automation

In order to speed up the screenshot and camera rotation process, we decided to add scripts to our camera object in the interior scene. One of Unity's main features as a game engine is the ability to add C scripts to objects to enable the objects in the scenes to perform actions for the user/player. Scripts can allow objects to move, perform their own actions, etc. To get an image of every aspect and angle of the objects we were performing the tests on, we decided to have the camera rotate around the object. This was accomplished using one of the aforementioned C scripts. [8]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CamRotation_for_camera : MonoBehaviour
{
    public GameObject Table;

    void Update()
    {
        transform.RotateAround(Table.transform.position, Vector3.up, 20 * Time.deltaTime);
    }
}
```

Fig. 7. Image the camera rotation script from Unity.

Important Unity built-in-classes of note utilized in the scripts for our project are [22]:

- Transform: Provides the script-writer with a way to work with the position, rotation, scale by manipulating the script. Also allows you to create parent and child game objects.
- Vectors: Built-in-class for the expression and manipulation of 2D, 3D, and 4D lines, directions, and points.
- Time: Class that allows you to measure and control the time in-game and manage the framerate of your game.
- Random: Allows the script writer to generate various common types of random values.

Using Unity's built in function "transform" and "rotate around" we set the transforming motion to be rotation around an object: the table in the center of the room in the scene. Through the script, we also set that the rotation is set to rotate based on the current time through the Update() method in the script/ The update method is triggered whenever the frame in a scene is updated. By having the rotation set this way, this allowed the desired rotating object (the camera) to rotate at a constant speed around the desired object of focus.

The actual screenshot action of the camera was also set by a C script. Similarly to the script for the camera rotation, the screenshot action declared by the script heavily relied on the variable of time. Through the script, we set the camera to take a screenshot every second the game was running. Before the actual taking of the screenshot, we had to make sure that the screenshots themselves were large enough for analysis and testing. In order to enable constant adjusting of the image size, we made the width and height of the image an adjustable field in the Unity editor, allowing us to simply type in the size, rather than comb through the code to find it.

After that, the steps to actually making the screenshot possible consisted of creating a new 2D texture (Texture 2D), have the texture read the current pixels of the screen and obtain their data (ReadPixels), encode those pixels to png

(EncodeToPNG), and then write that png and its bytes to a new file (WriteAllBytes). In addition to that, we had to define a save directory for all the resulting screenshots, and pass that off to the WriteAllBytes method as it was creating the file.

Later in the report, it is mentioned how motion blur is implemented for darker scenes in our lighting changes. This is because motion blur is a common occurrence in images taken in low-light settings. Unity allows users to apply motion blur through post-processing effects, but we ended up creating a motion blur effect by simply adjusting the speed of the camera during the auto-screenshot process. By having the camera spin at a speed 1.5x faster than Time (so a little faster than turning per second), we were able to have the motion blur effect we needed for certain settings. During these tests, we also increased screenshot frequency to be in proportion to that of scenes where motion blur was not necessary to allow for consistency in image angles/rotation in datasets among scene manipulations.

E. Textures/Materials

Because Unity is a game engine, it allows users to manipulate their objects in various ways. One of these important features is the ability to alter many different aspects of the objects in your scenes. For one of the tests in our study, we decided to see how the changing of an object's texture or material could affect the accuracy of YOLOv4's object recognition processes.

First, to alter the texture of the objects, we had to find different textures for the objects. We decided that the 3 most common textures for chairs are hickory, black plastic, and wooden (in addition to the green chair texture that came with the scene by default. Unity has an online free-market store filled with many assets (2D and 3D) for sale or for free for their creators, by creators. Through the store, we found the needed textures' png files.

In order to actually apply the texture to the object, we had to select the objects, locate them in the inspector of edit mode, and edit their albedo value in the main maps section of the inspector to obtain their source from the texture file we wanted.

F. Object Orientation

For one part of our experiment, we wanted to test YOLOv4's object recognition features under the circumstances of an object rotated about the origin in a way that would differ from how it normally would in reality. For this study, we decided to rotate only the chairs in the scene about their axis. Also, for this study, we wanted to test the accuracy of YOLOv4 in a scenario when the chairs are rotated backwards or forwards and the table and the bowl are still in their normal positioning.

For the non-random tests of YOLO's accuracy, we went through to Unity's inspector in their editor mode, selected the objects we wanted to apply the rotation to, and adjusted the values of their Transform aspect of the editor to rotate the object by a fixed amount while the camera turned around the table. For the fixed tests, we went through and adjusted the chairs by 10 degrees in both directions to test the accuracy of

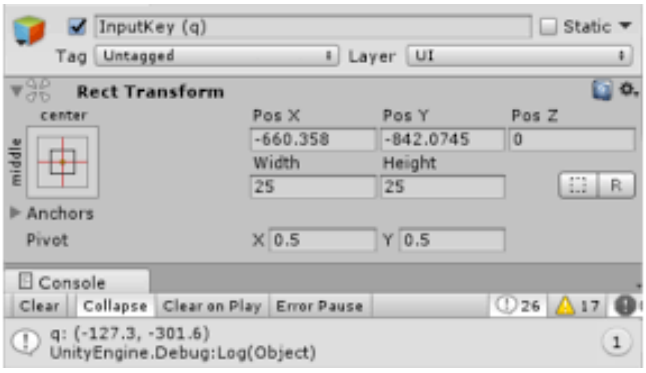

Fig. 8. The transform box of Unity, where object position can be manually manipulated.

YOLOv4 in the range of tilt from 180 to -180 degrees (with 90 degrees being the normal, upright position of the chairs.

For the tests where the chairs were rotated randomly, we enabled the rotation through a C script similarly to the script for the rotation of the camera. However, unlike the camera script, since the chair is rotating about itself rather than an object, we had to add extra methods and parameters to ensure the chair rotated in a way still visible to the camera. In the "start" function, which activates before the first frame update, we had to call the InvokeRepeating method on the "rotateAngle" function we created.

Unity contains a built-in factor of the built-in Vector3 class called eulerAngles (Vector3 are the 3D version of the built-in-class of Vector for Unity). Transform.eulerAngles is used to represent three-dimensional rotation in the scene. This rotation occurs as three separate rotations around individuals axes. In Unity, these are the Z axis, Y axis, and Y axis, in the order of which they occur. After receiving the values, they are converted to the Quarternion's internal format. In Unity, Quarternions are used to represent rotations. [8]

The rotateAngle function consisted of a Vector3 euler variable in which we called the transform method on the built-in eulerAngles definition in Unity. Then, with the euler variable, we called the x dimension of rotation of the object (x, y, z), and told it choose a random degree to turn to using the Random and Range methods in Unity (Random.Range(0f, 360.f)). Then, in order for the object to actually receive the value, we passed the transformed angles back in the entire euler value of the object.

G. Backgrounds

Another test we wanted to perform was to see how the difference in background could affect the accuracy of YOLOv4's object recognition feature.In order to change the background of the scene, we had to use many different methods. For the background of having no room in the scene and only showing the outdoors, we had to find the wall(s) object in the editor, and select it to be invisible. From then on, we continued with the screenshots.

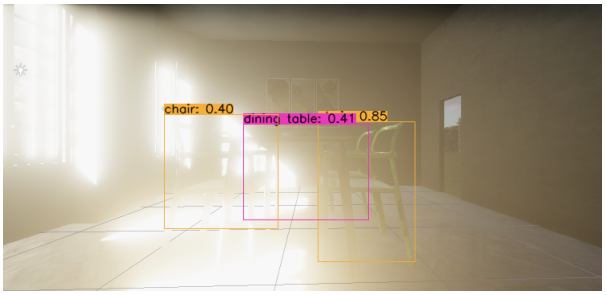

Fig. 9. Example of YOLOv4's findings in the 100,000 lux light setting.

Another aspect of the background we altered was the environment outside of the room. Normally, in the scene, the outdoor scenery is similar to that of a courtyard at a school. We were able to change this by changing the skybox of the scene. In Unity, a skybox is a cube-like object that appears behind all of the objects in a scene/game. In order to create the box, you have to create/select 6 textures to correspond to each side of the cube of the skybox. For our scene in particular, we took the normal skybox and made it a clear blue setting outside of the room to give the room a different background and even a different sense of lighting.

IV. Experiments and Results

For each experiment, we applied the specified changes we were testing, and then went through and tracked how often YOLOv4 was actually correct in its identification.

In each chart/table, we refer to the 6 objects as chair 1, chair 2, chair 3, chair 4, table, and bowl.

Referring to Figure 5:

- Chair 1 is the first chair on the left in the front facing view of the scene.
- . Chair 2 is the chair directly behind chair 1 on the same side of the table
- Chair 3 is the first chair on the right side of the table in the front.
- Chair 4 is the chair directly behind chair 3 on the same side of that table.
- Table is in reference to the chair in the scene.
- Bowl is referring to the bowl on top of the table.

A. Lighting Changes

We found it surprising that YOLOv4 (untrained) performed relatively "well"; (or normal) under extreme (low and dark) lighting conditions. We were expecting a huge drop off in accuracy in the dark and bright settings of the scene. There was even a slight increase in the performance of the algorithm during the bright sunlight scenes.

YOLOv4's object recognition algorithm performed just as well (and even slightly better) than it had in the normal lighting setting with an exception for a fefw objects.

YOLOv4's object recognition algorithm performed as expected under the normal lighting setting of the scene.

Fig. 10. "Normal" Lighting results after being run through the YOLOv4 process.

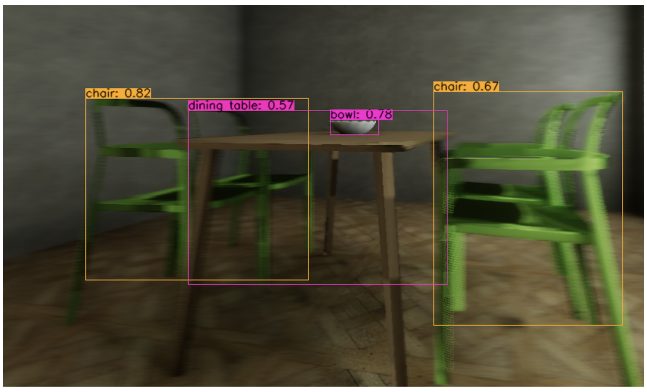

Fig. 11. Example of YOLOv4 results with low lighting settings and motion blur.

YOLOv4's object recognition algorithm performed just as well than it had in the normal lighting setting with the exception of a few objects.

Lighting Changes Chart (Out of 60 images for each lighting setting)

Objects	Lighting Settings		
	Dark Accuracy (0 lux)(%)	Normal Accuracy (20,000 lux)(%)	Bright Sunlight Accuracy (100,000)(%)
Chair 1	55.55	52.94	64.71
Chair 2	50	41.18	52.94
Chair 3	38.88	58.82	70.59
Chair 4	61.11	58.82	58.82
Table	61.11	64.71	64.71
Bowl	61.11	58.82	52.94

Fig. 12. YOLOv4 accuracy when the lighting (lux values) were manipulated.

The dataset consisted of 60 images for each light setting.

There were slight increases and drops in accuracy here in there for some objects in some scenes, but that could be due to camera rotation and screenshot slight differences and the size of the data set between each category.

B. Positioning Changes

As expected, YOLOv4's object recognition algorithm was very inaccurate when it came to the rotation and positioning of objects in the scene whenever these variables were out of the range of normal.

POSITIONING CHANGES CHART (OUT OF 60 IMAGES FOR EACH POSITION SETTING)

Objects	Positioning Changes		
	Normal Positioning Accuracy (%)	**Placed Positioning (Forwards and Backwards) Accuracy (%)**	**Random Positioning Accuracy (%)**
Chair 1	52.94	13.89	38.24
Chair 2	41.18	11.11	14.71
Chair 3	58.82	16.67	32.35
Chair 4	58.82	19.44	11.76
Table	64.71	22.22	70.59
Bowl	58.82	30.56	76.47

Fig. 13. YOLOv4 accuracy when the position of the chairs (rotation values) were manipulated.

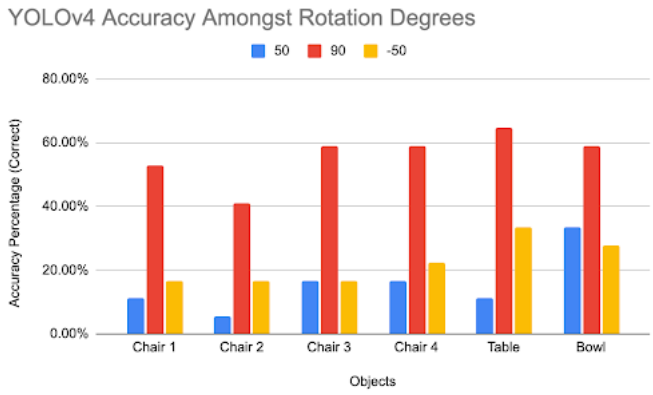

Fig. 14. YOLOv4 accuracy when the position of the chairs (rotation values) were manipulated - further explored.

Fig. 15. An example of when YOLOv4 would incorrectly identify one of the chairs as something other than a chair.

The dataset consisted of 60 images for each manual rotation value, except for the placed positioning values, which were added together and averaged out (30 for 50 degrees, 60 for 90, and 30 for -50) and 30 images in total for the random rotation dataset.

We observed that when we rotated the chairs backwards and forwards, YOLOv4 had a very hard time with recognizing each object. With slight tilts of up to about 30 degrees forwards and backwards, YOLOv4 could at least recognize some of the normally positioned objects and around 1-2 of the tilted objects (the chairs). However, once we rotated the chairs past 30 degrees, YOLOv4 would fail to recognize any of the objects in the scene or guess the identity of the object incorrectly.

Another interesting phenomenon is that YOLOv4 even had difficulty recognizing non-rotated or "normal" objects whenever the objects surrounding them were rotated in an abnormal position.

Also, the values of 50 degress were used as a baseline rotated factor for our results, because we noticed that once the chairs weer rotated past 50 degrees forwards or backwards, none of the objects in the scene were recognized by YOLOv4.

Common incorrect guesses for the objects included "vase" or "cup" for the bowl object in the scene and "potted plant" for one of the chairs in the scene.

C. Material Changes

MATERIAL ACCURACY CHANGES CHART (OUT OF 33 FOR EACH MATERIAL)

Objects	Positioning Changes			
	Normal (Green Plastic) Accuracy (%)	**Wooden Accuracy (%)**	**Black Plastic Accuracy (%)**	**Hickory Chair Accuracy (%)**
Chair 1	52.94	51.51	60.60	60.60
Chair 2	41.18	48.48	51.51	60.60
Chair 3	58.82	57.57	48.48	72.72
Chair 4	58.82	63.63	54.54	69.69
Table	64.71	60.60	84.84	78.78
Bowl	58.82	57.57	66.66	96.96

Fig. 16. YOLOv4 accuracy when the material of the chairs (texture files and material values) were manipulated.

The dataset consisted of 33 images for each material (33 for black, wooden, hickory, and normal, each).

Similarly to the tests/experiments with the changing of the lighting settings in the interior setting in Unity, the results of YOLOv4's object recognition process did not differ much when we changed the "material" of the chair. In addition to that, we also found that there was no observable difference in YOLOv4's accuracy amongst the different textures as well – for example, the black plastic material made the chair just as recognizable to the DNN as the green plastic material. Despite the darkness of the black chairs, YOLOv4 still found them as easily recognizable as before.

Notable findings included the strange phenomenon amongst the test for the hickory chair materials: sometimes YOLOv4 would detect a dog amongst the shadows or bright spots of

Fig. 17. Example of YOLOv4 accuracy when the chairs had the black plastic material applied to them.

lighting in the test images. This did not occur in any of the other tests for the different materials for the chairs in the scene. In addition to the appearance of the dog, there was a slight increase in accuracy off YOLOv4's object recognition processes when the chairs had the hickory material applied to them.

There were slight increases and drops in accuracy here in there for some objects in some scenes, but that could be due to camera rotation and screenshot slight differences and the size of the data set between each category.

D. Background Changes

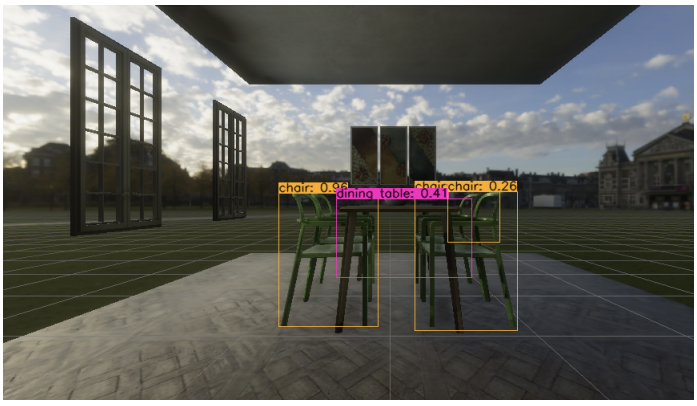

Fig. 18. YOLOv4 accuracy when there are no walls in the scene.

The dataset consisted of 33 images for each background change (33 for no walls, blue and normal, each).

Similarly to the results for the tests with the lighting, there was no notable difference for the accuracy of YOLOv4's object recognition process in scenes with an altered background versus that of the original background that came with the scene. There were slight increases and drops in accuracy here in there for some objects in some scenes, but that could be due to camera rotation and screenshot slight differences and the size of the data set between each category.

Objects	Background Changes		
	Normal Positioning Accuracy (%)	No Walls Background Accuracy (%)	Blue Outside Background (%)
Chair 1	52.94	51.51	60.60
Chair 2	41.18	42.42	39.39
Chair 3	58.82	57.57	60.60
Chair 4	58.82	60.60	54.54
Table	64.71	60.60	57.57
Bowl	58.82	54.54	51.51

Fig. 19. YOLOv4 accuracy results with background changes to the scene.

We were surprised to see virtually no drops in accuracy in the scenes where there were no walls. Once the walls were removed, there was a lot of "noise" in the background of the scene. The default background, as stated before, appears to be. the front yard/courtyard of a school, with a lot of trees, contrast in colors, buildings, and other unidentifiable objects in the background. We thought this would have resulted in YOLOv4 having less of a easily identifiable way to find the clear beginning and end of the objects in the scene.

E. Skull Results

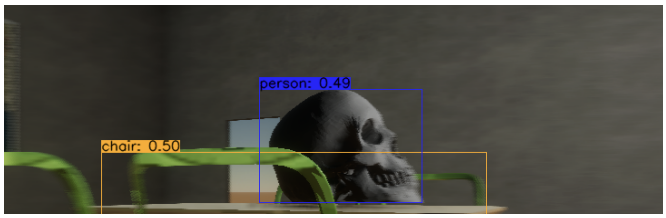

Fig. 20. An example of YOLOv4's guesses at the identity of the skull.

The dataset consisted of 33 total images of the skull object placed on the table in Interior Rendering Tutorial scene.

A chart was not included for YOLOv4's accuracy for the skull because YOLOv4's object recognition algorithm never guessed the identity of the skull correctly. Some of the algorithm's guesses for the object included:

- bird
- an additional chair in the scene
- vase

The guess for the identity of the skull was either incorrect or YOLOv4 did not recognize the skull as an object at all. The most accurate guess for the skull was "person".

V. Discussion and Conclusions

A. Shortcomings

Unity is a game engine more so known for their 2D or cartoon-like 3D graphics, not for their photo-realistic 3D graphics. Whether in editing mode or in game mode, the scene always looked somewhat pixelated - not completely real. In the future, a better game engine to conduct this study with would

probably be Unreal Engine by Epic Games. Unreal Engine, unlike Unity, is known for its powerful 3D graphics rendering engine, and it is the go-to engine for photo realistic games by many industry leaders [19].

Fig. 21. YOLOv4 accuracy when the position of the chairs (rotation values) were manipulated.

Although Unity does have a large community of developers, Unreal Engine has a very large and very active community of members who are constantly posting questions, answers, and content for other users. Because of this, issues may have been easier to resolve due to the increased likelihood of finding help or coming across a previous asked question similar to our issues in Unreal's developer community forums (posts appear as often as multiple times per hour).

Another downside to the Unity vs Unreal debate is that having access to Unity's high-end art asset pack relies on purchasing at least Unity Plus, which will require you having to spend at least $399 per year. Gaining access to Unreal's high-end assets is completely free - there is only one version off Unreal, and it is completely free. Certain features that would be helpful to the study are also more difficult to do in Unity than Unreal Engine.

One of the tasks we originally set out to do was to find the ground truth of all of the objects in our interior scene. We were hoping to accomplish this through using the Depth Processing features of Unity. We were able to find tutorials online that worked for some, but not others, and the process of finding the depth of objects is known to be difficult in Unity.

Differences between Unity updates and versions caused discrepancies in code that would work in some cases and then not work in others. We noticed that the tutorials we found were effective for older versions of Unity, but the tutorials were not effective for newer versions of Unity that we tried the tutorial on. Because Unreal is commonly used for very realistic and accurate 3D scenes, they have made the task of finding the depth values and ground truth of objects much easier for their users [9].

Another short comings of the study is the lack of results and information about the process of training a custom dataset to recognize the rotate objects, but we will discuss that more in the paper.

Another aspect of the study we would like to further explore or change is the testing of the backgrounds and YOLOv4's object recognition accuracy. The only backgrounds possible/chosen were the backgrounds with no walls and the background in which the outdoor scenery behind the walls was changed. In our opinion, the aspect of the background affecting the accuracy was probably not fully explored due to the limiting amount of options. In the future, we would like to explore the possibility of changing the colors of the walls and adding a patterned texture to the walls.

In addition to the background, we also did not get the chance to study how the arrangement of objects could have affected the accuracy of YOLOv4. For the entirety of the study, we kept the scene in the exact same arrangement in order for it to have a constant value connect each test in the study. However, it may have been interesting to keep the original scene as a comparable value to scenes with the tables in abnormal positions. We could have placed the chairs on the table, placed the bowl on the chair, etc., because these arrangements of these objects is possible in reality. If there was a difference in accuracy amongst certain locations of objects, that would have definitely something notable to track.

An additional shortcoming in the study is the size of the datasets. We were able to automate the process of object rotation, camera rotation, screenshots, and even running YOLOv4's object detection on images, we were not able to automate the process of checking whether or not YOLOv4's identification was correct or not. In order to ensure the objects were correctly identified, we had to go through every YOLOv4 labelled photo and take note of whether. the object was labelled, and if so, whether the label was actually correct. Because this process was manual, it limited the size of the dataset for us.

B. Discussion and Future Directions

Overall, we found that YOLOv4, before the training process can be very flawed in the object detection process for objects that are varied even slightly differently than normal. This, when applying the findings to real-implications, is slightly concerning. Although Yolov4 did well in the lighting settings - one of the main contributors to the accident resulting in the loss of life of Herzeburg - the failures of the DNN in situations where the chairs could be applied to scenarios in the real world.

One interesting finding across every single test for YOLOv4, the windows in the scene and the paintings in the scene were never identified by the YOLOv4 object recognition algorithm. In reference to a topic discussed later on the paper, this could be due to the absence of windows or that specific painting in the dataset created by the photographers who made and labelled the original dataset.

The constraints of the original training dataset for YOLOv4's object recognition algorithm could be a potential reason for its failure to recognize the skull object when placed it in the scene.

On the road on sharp turns, sometimes cars can rotate a bit and angle upwards towards the air. If YOLOv4 were being used as the objection recognition algorithm for some self driving cars, then the car utilizing it would fail to recognize that slightly turned car in front of them. This would result in an accident. There are plenty of real-life scenarios involving object positioning in addition to the previous one that explain how critically important objection recognition DNN accuracy is.

In the future we are hoping to test YOLOv4 in even more varying scenes or manipulations. For example, another parameter of the scene we want to manipulate is the transparency of objects. We also want to test the effect of camera vertical position on the accuracy of YOLOv4 in recognizing objects. We are also hoping to recreate some of these tests and processes with multiple photo-realistic scenes (not just the interior scene), objects, and materials.

In addition to that, we are hoping to see how the training process of DNNs can affect the accuracy of them in the abnormal settings we tested for today. One of YOLOv4's most well-known features is its ability to allow the user to train the DNN to recognize their own custom objects. The original data. we gathered utilized the dataset and labels that came with the original installment of YOLOv4. However, the training process is known to be difficult, but the developers of the YOLO series have been trying to work on it.

As mentioned before, the absence of a skull label in the original training dataset for YOLOv4's object recognition algorithm may have been why YOLOv4 struggled with identifying our skull object. Because YOLOv4's training process allows for the creation of custom object labels, we could try training YOLOv4 on the skull object. After training, we could see if the failures of YOLOv4 in recognizing the skull was due to accuracy of YOLOv4 or if it is due to YOLOv4 not being fed information about skulls in its original creation and training.

REFERENCES

[1] AlexeyAB, "AlexeyAB/darknet," Darknet. [Online]. Available: https://github.com/AlexeyAB/darknet. [Accessed: 11-Nov-2020].

[2] U. Wolz, G. Carmichael, and C. Dunne, "Learning to Code in the Unity 3D Development Platform," Proceedings of the 51st ACM Technical Symposium on Computer Science Education, 2020.

[3] hunglc007, "hunglc007/tensorflow-yolov4-tflite," GitHub. [Online]. Available: https://github.com/hunglc007/tensorflow-yolov4-tflite. [Accessed: 11-Nov-2020].

[4] J. McCormac, A. Handa, S. Leutenegger, and A. Davison, "SceneNet RGB-D: 5M Photorealistic Images of Synthetic Indoor Trajectories with Ground Truth," Dec. 2016.

[5] M. A. Alcorn, Q. Li, Z. Gong, C. Wang, L. Mai, W.-S. Ku, and A. Nguyen, "Strike (With) a Pose: Neural Networks Are Easily Fooled by Strange Poses of Familiar Objects," 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Nov. 2018.

[6] R. Jones, "Using physically-based lighting values with Enlighten and UE4: Enlighten - Real Time Global Illumination Solution: Silicon Studio," Using physically-based lighting values with Enlighten and UE4 — Enlighten - Real Time Global Illumination Solution — Silicon Studio, 22-Mar-2019. [Online]. Available: https://www.siliconstudio.co.jp/middleware/enlighten/en/blog/2019/20190322/. [Accessed: 11-Nov-2020].

[7] R. Randazzo, "Who was really at fault in fatal Uber crash? Here's the whole story," The Arizona Republic, 18-Mar-2019. [Online]. Available: https://www.azcentral.com/story/news/local/tempe/2019/03/17/one-year-after-self-driving-uber-rafaela-vasquez-behind-wheel-crash-death-elaine-herzberg-tempe/1296676002/. [Accessed: 09-Nov-2020].

[8] U. Technologies, Unity User Manual (2019.4 LTS), 27-Oct-2020. [Online]. Available: https://docs.unity3d.com/Manual/index.html. [Accessed: 09-Nov-2020].

[9] Unreal Engine 5. [Online]. Available: https://fasraspen586.weebly.com/unreal-engine-5.html. [Accessed: 11-Nov-2020].

[10] L. Liu, H. Li, and M. Gruteser, "Edge Assisted Real-time Object Detection for Mobile Augmented Reality," The 25th Annual International Conference on Mobile Computing and Networking, 2019.

[11] W. Qiu, F. Zhong, Y. Zhang, S. Qiao, Z. Xiao, T. S. Kim, and Y. Wang, "UnrealCV," Proceedings of the 2017 ACM on Multimedia Conference - MM '17, 2017.

[12] J. Choi, H. J. Park, J. Paek, R. K. Balan, and J. G. Ko, "LPGL: Low-power graphics library for mobile AR headsets," Yonsei University, 12-Jun-2019. [Online]. Available: https://yonsei.pure.elsevier.com/en/publications/lpgl-low-power-graphics-library-for-mobile-ar-headsets. [Accessed: 14-Nov-2020].

[13] X. Ran, C. Slocum, M. Gorlatova, and J. Chen, "ShareAR," Proceedings of the 18th ACM Workshop on Hot Topics in Networks, 2019.

[14] "Unity vs Unreal: Ultimate Game Engine Showdown," The Ultimate Resource for Video Game Design, 12-Jun-2020. [Online]. Available: https://www.gamedesigning.org/engines/unity-vs-unreal/. [Accessed: 15-Nov-2020].

[15] "The most powerful real-time 3D creation platform," Unreal Engine. [Online]. Available: https://www.unrealengine.com/en-US/. [Accessed: 15-Nov-2020].

[16] U. Technologies, Unity. [Online]. Available: https://unity.com/. [Accessed: 15-Nov-2020].

[17] "Microsoft HoloLens: Mixed Reality Technology for Business," Microsoft HoloLens — Mixed Reality Technology for Business. [Online]. Available: https://www.microsoft.com/en-us/hololens. [Accessed: 15-Nov-2020].

[18] "Augmented Reality," Information Management and Decision Architectures. [Online]. Available: https://www.nrl.navy.mil/itd/imda/research/5581/augmented-reality.

[19] Paul Gestwicki. 2019. Unreal engine 4 for computer scientists. J. Comput. Sci. Coll. 35, 5 (October 2019), 109–110.

[20] A. Bochkovskiy, C.-Y. Wang, and H.-Y. Mark, "YOLOv4: Optimal Speed and Accuracy of Object Detection," 2020.

[21] J. Nelson, "Responding to the Controversy about YOLOv5," Roboflow Blog, 05-Oct-2020. [Online]. Available: https://blog.roboflow.com/yolov4-versus-yolov5/. [Accessed: 15-Nov-2020].

[22] U. Technologies, "Important Classes," Unity. [Online]. Available: https://docs.unity3d.com/Manual/ScriptingImportantClasses.html. [Accessed: 15-Nov-2020].

[23] R. Böhringer, "Postprocessing with the Depth Texture," Ronja's Shader Tutorials, 30-Jun-2018. [Online]. Available: https://www.ronja-tutorials.com/2018/07/01/postprocessing-depth.html. [Accessed: 15-Nov-2020].