LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT
School of Energy Systems
Mechanical Engineering Programme

Jacob Schubbe

OPTIMAL VIEW ORIENTATION OF THREE-DIMENSIONAL PARTS AND CUSTOMIZABLE CAMERA PATH – UTILITIES FOR UNITY

Lappeenranta, 16 November 2020

Examiners: Professor Aki Mikkola
Adam Klodowski

ABSTRACT

Lappeenranta-Lahti University of Technology LUT
School of Energy Systems
Mechanical Engineering Programme

Jacob Schubbe

Optimal View Orientation of Three-Dimensional Parts and Customizable Camera Path – Utilities for Unity

Master's Thesis

2020

97 pages, 107 figures, 7 tables, 7 equations, and 2 appendices

Examiners: Professor Aki Mikkola
Adam Klodowski

Keywords: view orientation, camera path, utility, Unity Engine, MOSIM, machine learning

Simulations for the assembly of a product in a manufacturing plant plays a critical role in the successfulness of the product and the company. For the project MOSIM, Unity is used for completing this simulation work. When defining the tasks that need to be accomplished during the simulation, the possible parts from which the user can choose need to be displayed in such a way that the user is able to select the part based solely on the image because in some cases, the name of the part is not descriptive enough. This optimal view of the part can be determined by measuring certain metrics from the views of the object. These metrics are compared by using one of two methods: Weighted Scores or *k* Nearest Neighbors (*k*-NN).

In addition, during the simulation of the assembly, manual movement of the camera was previously used but became tedious to do every time the simulation was run. A new camera path utility was created which allowed the user to manually and intuitively define specific camera points that could be saved and loaded to make a complete camera path through which the camera would move during the simulation. Both goals were met and successfully implemented into Unity. Through testing and a survey conducted, the weighted scores currently result in better views of the tested objects as the optimal view from the weighted scores tend to show most of the features of the object. On the other hand, *k*-NN produced less than optimal views for the objects because some of the views were not showing enough detail of the object. *k*-NN is considered the preferred method, though, as the method allows the results to improve overtime as the training data set increases in size.

ACKNOWLEDGEMENTS

Firstly, I would like to express my thanks to both of my advisors of my thesis work, Aki Mikkola and Adam Kłodowski, who assisted and guided me through the thesis process for the last six months and who gave me an opportunity to work on a project in a field that interested me.

In addition, I would like to thank my Fulbright advisor at the University of Maryland Baltimore County (UMBC), Brian Souders, and one of my professors from UMBC, Marc Zupan, for the help that they provided when I was applying for my Fulbright Grant to study in Finland. Without them, I would not have been able to earn a Fulbright Grant to study for my master's degree abroad. I would like to thank the Fulbright Finland Foundation and Lappeenranta-Lahti University of Technology (LUT University) as well for the opportunity that they gave me to come to Finland for my studies.

Also, appreciation is deserved by my peers in my courses and the Aalto family, who is my Finnish family from the "Meet a Local Family" program. They all accepted me into their inner circles and showed me more about their traditions and cultures, and for that, I could not be more appreciative.

Finally, I would like to thank my parents, friends, and family for the love and support that they have shown me, not only during my master's degree but also throughout all the years of my life. Without your constant support, I would not be where I am today, and I am forever grateful.

Thank you!

-Jacob

TABLE OF CONTENT

APPENDICES

Appendix I: Training Data Points and Plots for *k*-NN Algorithm

Appendix II: Survey of the Best Orientation between Procedures

LIST OF SYMBOLS AND ABBREVIATIONS

Symbols:

a	Number of points for the class c in the *k*-NN region
c	Class for *k*-NN, in this case the selected metric
D_{ij}	Distance from the new point, j, to the reference point, i [units of Unity]
i	Reference point for *k*-NN
j	New point for *k*-NN
k	Number of surrounding points used for *k*-NN algorithm
n	Number of dimensions for the distance D_{ij}
m	Current 'dimension' (metric) being analyzed
p	Number of pictures for a specific rotation direction
$\Delta\varphi$	Angle increment of rotation for roll [degrees]
$\Delta\psi$	Angle increment of rotation for yaw [degrees]
r	Ratio of visible triangles to total triangles
T	Total number of triangles in the mesh
T_c	Total score for class c
$\Delta\theta$	Angle increment of rotation for pitch [degrees]
v	Number of visible triangles in the mesh
x	List of values for a specific metric
x_i	Individual value from a list x
$x_{i_{norm}}$	New normalized value based on x_i
x_{vi}	Position of the reference point i for the n^{th}-dimension
x_{vj}	Position of the new point j for the n^{th}-dimension
Z	Number of different "best views" that the user defined

Abbreviations:

3D	Three-Dimensional
API	Application Programming Interface
AI	Artificial Intelligence
GDC	Game Developer Conference
k-NN	*k*-Nearest Neighbors

ML	Machine Learning
MMI	Motion Model Interface
MMU	Motion Model Unit
MSD	Musculoskeletal Disorder
RGBA	Red-Green-Blue-Alpha
R&D&I	Research, Development, and Innovation
XML	Extensible Markup Language

1 INTRODUCTION

In the mechanical industry, a common method for completing assembly work is by building or producing a product in such a way that the assembly moves to varying locations in the manufacturing facility, which allows workers to complete a particular job on the assembly at that location (Cambridge University Press, 2020). Manufacturers want to deliver quality products to customers which means assembly efficiency and accuracy must be maintained at a high level in order to avoid errors in the final product (Falck, et al., 2012). In assembly work, there is an operator choice process and complexity which affects how the assembly task can get done, especially if there is customer customization in the order, as they have to choose the correct part from all the possible parts to include. In complex assemblies, it may even be required from the operator to make choices under time constraints and pressure, such as picking the right material, tool, or method. (Falck, et al., 2012) An example of this can be seen in **Figure 1** and **Figure 2**, where the operators need to know which parts to pick for certain products. Some assembly processes may contain products that have many parts from which to choose and assemble, like in **Figure 1**, while other assembly lines may have different models with customization options, as seen in **Figure 2**.

Figure 1: Choosing the correct part from many options (Weber, 2016)

Figure 2: Mixed-Model assembly with similar family of products but possible customization (Weber, 2016)

In situations such as these, automation can become particularly useful. With the introduction of new technology, skilled human operators can ensure that the production has increased efficiency and rates by using modern automated production processes. The range of

automation ranges from manual, to semi-automated, to automated. For a manual process, this usually means the human operators complete the whole task. In semi-automated systems, the systems complete alignment and specific tasks while human operators handle the material. In fully automated systems, there is little to no interference by human operators as the material handling and tasks are completed by the robot. These automated systems are important especially with constrained and repetitive work with less opportunities for short breaks during work, as this can lead to the development of musculoskeletal disorders (MSDs), especially in the neck, shoulders, hands, and upper back. (Locks, et al., 2018)

Although some assembly work is able to be automatized and thereby increasing the efficiency of the assembly process, due to some manufacturing plant layouts and tasks that need to be accomplished, it is possible that manual assembly is still required. The efficiency of manual work can be improved using computer aided planning and human motion simulation. Computer aided planning, in addition to digital planning tools, need to be incorporated into the planning process to be more efficient and competitive due to the growing number of competitors. By including three-dimensional (3D) human models and simulations, it is possible to obtain objective, repeatable results through realistic and risk-free simulations. This, in turn, allows for faster iterations in planning and reduced costs. In addition, there would be no need for manual testing or physical hardware as the simulation would be able to analyze hypothetical scenarios and present comprehensive assessments and optimization based on varying metrics. (Gaisbauer, 2019)

Human motion simulation can be applied in many different fields including manufacturing (e.g. work cell layout, workflow simulation, safety analysis, lifting), but could also include product design (e.g. comfort, visibility, multi-person interaction), motion analysis and medical applications (e.g. highly detailed musculoskeletal simulations, detailed analysis of underlying processes, sports), infotainment and human-computer interaction (e.g. intuitive interaction with digital representation, maximizing user experience), and entertainment/gaming (e.g. creation of lifelike characters, movies, gaming, advertisements). Although human motion simulations have been utilized previously for research, industry, and entertainment, there is no framework available which incorporate all needs into one utility, such as picking up a tool (motion blending), walking (machine learning), and entering

a car and fastening bolts (physics based), which all could use different coding languages depending on the task. This framework is especially needed in manual assembly, which is mainly comprised of multiple heterogeneous tasks and motions that require different assessment metrics and facets. Most simulation tools currently available only focus on one aspect such as buildability, ergonomics, etc. This means that there was a door left open for a utility that incorporates all the required aspects. (Gaisbauer, 2019)

MOSIM is an ITEA 3 project, which consists of transnational and industry-driven research, development, and innovation (R&D&I). The MOSIM project primarily focuses on "modular simulation of natural human motions" (Gaisbauer, 2019). The goal of MOSIM is to fill this opening with a new framework, which can help industries who face manufacturing challenges. The automotive assembly planning industry is one example of this. Most production plans that are currently produced fail to truly embody the manufacturing process and cause more energy to be exerted than anticipated. Third party providers have tried to create frameworks that can accomplish similar ideas but porting efforts for new technology is costly. In addition, there is a lack of knowledge and open source code for technology, so creating frameworks for this purpose has not been realistic. This is where MOSIM comes into play. By tackling these issues, an idea for a tool can be envisioned to allow for improved planning of manual assembly processes. The idea for MOSIM is to help developers by establishing a standard for exchanges of digital human simulation, creating a framework that combines heterogeneous systems, and uses a target engine such as Unity to merge different approaches. (Gaisbauer, 2019)

1.1 Objectives

Once the framework is created, it is then possible to start to use a generic scenario. For the "Pedal Car" use case, the objective of the simulation is to confirm the feasibility and sequence of defined activities. During the Pedal Car use case, multiple inputs are provided: product structure (including all the parts), the cycle time and required number of stations (in this case, four stations with a cycle time of six minutes), manual assignment of parts to each station, the creation of a "high level" task list with the estimated assembly times, and the preparation of the simulation scene. (Gaisbauer, et al., 2020)

While defining the task list, the user needs to select the parts, the tasks to be completed and the order of the tasks. For simple cases, where the number of parts is limited, selecting parts based on their names is feasible, but for cases which include large assemblies, part names might not be clear enough to uniquely identify parts that should be used. For this reason, visual representation of the parts will be provided in the web-based task editor. To provide these visual representations to the task editor, a script needs to be created that will take photos of all the parts in the assembly from the scene in a virtual "photobooth." Semantic information allows for the identification of 3D objects in the scene, which are labeled as either a part, a tool, or it does not have a label attached which means it is simply scenography. This semantic information is obtained from the "MMISceneObject" script attached to each object. The MMISceneObject is a vital part of the MOSIM framework, which allows for the addition of meaning and function for 3D geometry in the scene. During this virtual photobooth, the part needs to be isolated from the other parts and needs to be oriented in its "best view," where 'best' means the view that provides the most visual information about the part.

In addition, during the simulation, the user should have the ability to define locations and orientations for the camera at key time instances in the simulation. This allows the camera to automatically move during the simulation rather than having the user manually move the camera during each simulation replay. It will also enable creation of videos from the output of the simulation, which can be directly used for training or marketing purposes. For this research, the pedal car will be the test use case and specific methods as to how these tasks will be accomplished will be further explained in Sections 3 and 4.

2 MOSIM PROJECT

Before explaining how these objectives will be accomplished, it is necessary to provide some background about the target engine used for the MOSIM project. The target engine is software that can perform 3D visualization in combination with the MOSIM framework tools. These tools enable communication between modules of the framework, control of the simulation, preparing and reviewing the simulation scene, and completing basic post-processing of the scene. There are many common platforms in use today that can be used as target engines for the MOSIM framework such as Unity, Blender, and Unreal Engine, but because the first, and most complete, framework implementation supports Unity, this research work will also utilize Unity 2018, whose scripting language is C# (C-sharp) (Microsoft, 2020). The core platform of Unity allows for real-time 3D development by artists, designers, and developers. The platform is typically used for creating games using the provided tools, which allow for the creation and publishing of games to a wide range of devices. Unity allows for real-time 3D creations and previewing with rapid editing and iterations in the development cycle. It is also possible to ”create once, deploy anywhere,” which means that after creating the game, Unity provides a simple tool for porting the game to varying devices such as augmented and virtual reality, mobile, desktop, console, and the web. (Unity Technologies, 2020)

2.1 History of Unity

The history of Unity has been long, with the original three developers Nicholas Francis, Joachim Ante, David Helgason starting the work in Copenhagen and starting the company as ”Over the Edge Entertainment” with Helgason taking the position as CEO. After spending about five months creating a game called *Gooball* and working out bugs, annoyances, and errors in this new engine, it was not long until the initial refined 1.0 version of Unity was released in June 2005. At this point in time only hobbyist and independent developers truly used Unity, as Unity version 1.0 was only available initially for projects that were going to be run on Mac OS X. (Haas, 2014)

In Unity version 1.1, Microsoft Windows and web browser support for projects was added, which was a large step up from the older Adobe Flash-based browser games, as Unity was using accelerated 3D graphics. Support for C/C++ plugins was also added, allowing software and hardware that were not initially supported out-of-the-box to be used. In 2007, Unity version 2.0 was released during the Unity Developer Conference. This conference was used for incorporating support for DirectX (Windows software) instead of OpenGL software, which had to be installed independently. This allowed Unity to enhance Windows and web player compatibility. (Haas, 2014).

About a year later in December 2008, a separate product known as Unity iPhone was marketed, which allowed for distribution to iPhones as well. This was necessary because of the increasing popularity of smart phones and iPhones at the time. Although Unity was capable of exporting projects for multiple systems now (e.g. Mac OS X, Windows, web browsers, and the iPhone), the editor for Unity was only able to run on Macintosh, but Windows was necessary as well. To fix this, at the 2009 Game Developer Conference (GDC), Unity version 2.5 was released. By September 2010, Unity version 3.0 was released and included many desired features requested by users, but most importantly, unified its different external editors (e.g. iPhone, Wii, etc.). At this point, Unity had many registered developers, was used frequently for educational purposes, and was also used often for mobile platforms. Unity version 4.0 was released later with other add-ons and support being added, while version 4.3 was released in November 2013 with 2D support, sprite support, and 2D game development options. More versions have been released since, but this gives a brief history of how long Unity has been improving and who the initial target audience of the software was. (Haas, 2014)

2.2 Additional Purposes of Unity

Although Unity's original intention throughout its history was for game development, the fact that now the engine (the editor and the systems on which the projects run) is capable of being used on multiple operation systems (Mac OS X, Windows, Android, iPhone, etc.) and is improving with more add-on support shows that the system's potential for different uses is expanding. Creating projects that represent real-world environments is possible through Unity because of the application programming interfaces (APIs) DirectX and OpenGL. Both

APIs have highly optimized rendering graphics pipelines which, in combination with the built-in physics engine, allows for real-world simulations. (Yang & Jie, 2011, p. 976)

In addition, Unity offers another form of software called Unity Simulation. With Unity Simulation, it is possible to run millions of occurrences of a specific Unity build by using cloud computing power. The project can be parameterized so it will change between occurrences of the build and output specific data. This resultant data could be used for machine learning purposes to train artificial intelligence (AI) algorithms or to evaluate and improve modeled systems like the MOSIM project. (Unity Technologies, 2020)

2.3 MOSIM Framework Integration

The MOSIM framework can be broken down to a few different layers of components. These components can be seen in **Figure 3**.

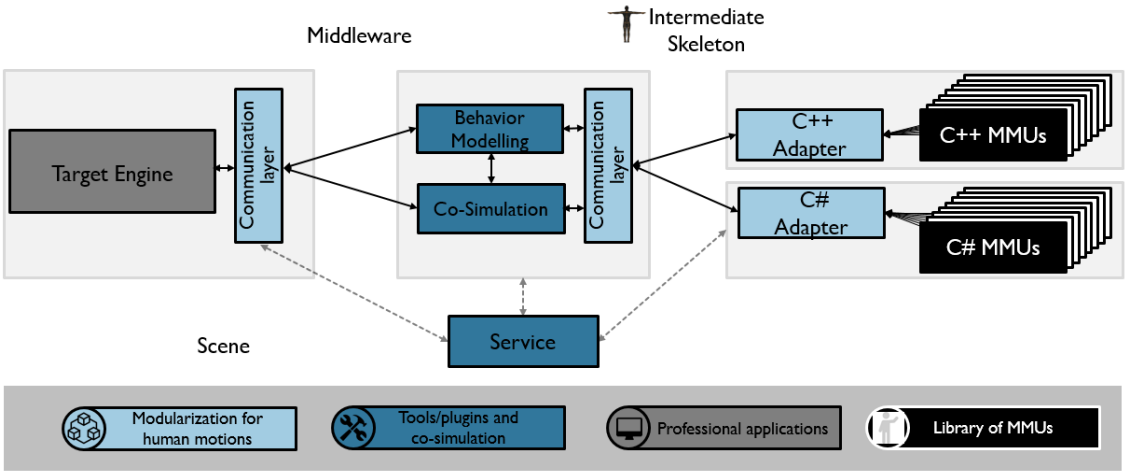


Figure 3: Framework Diagram with Key Components (Handel, 2020)

Starting on the left, as previously mentioned, the "Target Engine" used in this research is Unity because the first and most complete MOSIM framework is built for Unity. This framework consists of a few different components. Starting from the basic building blocks on the right side of **Figure 3**, the Motion Model Units (MMU) allow for the embedding of various animations for certain tasks within the framework (Gaisbauer, et al., 2019). These animations include reaching, walking, grasping, turning, etc. (See **Figure 4**.) These MMUs

can then be sent through an adapter which allows for the integration in other programming languages and eventually other target engines as well. After the adapter, the framework has two utilities: Behavior Modeling, which includes discrete MMU sequences for certain tasks, and the Co-Simulation, which allows for continuous movement of the Skeleton throughout the simulation using various Motion Model Interfaces (MMI) as seen in **Figure 5**. (Handel, 2020) Finally, these sequences and movements are transferred through to the target engine, where the user will be able to see the actions happening. Meanwhile, the "Services" block in **Figure 3** is used for certain actions that run in the background of the framework.

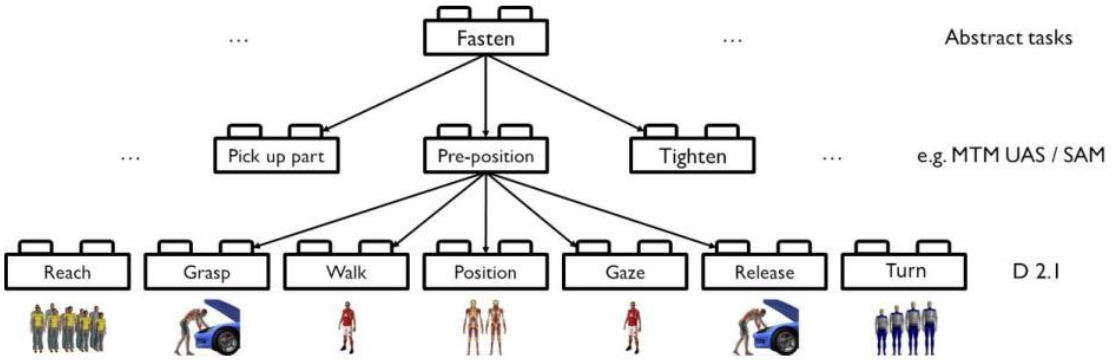


Figure 4: Task and MMUs (Handel, 2020)

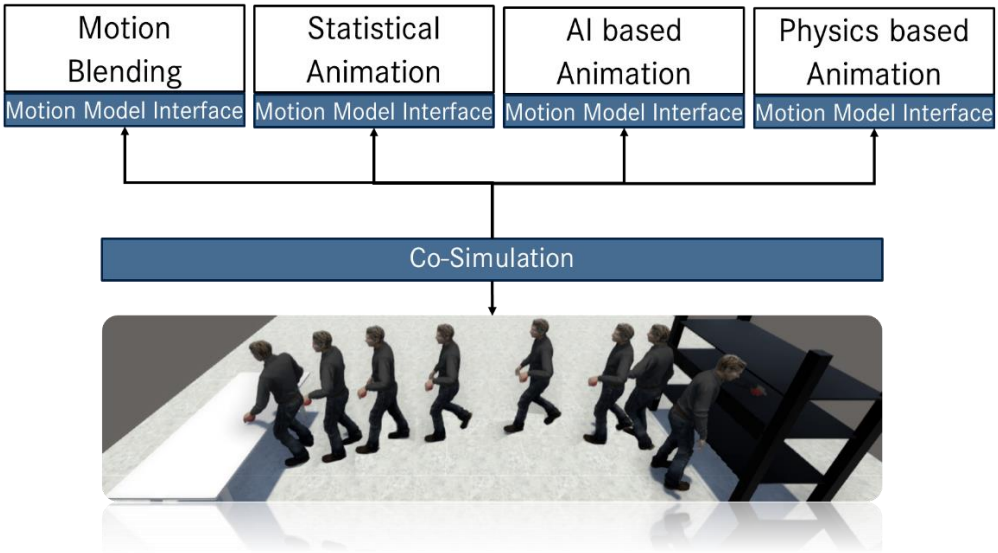


Figure 5: MMI Framework for Co-Simulation (Handel, 2020)

2.4 Unity Editor

The Unity editor has two modes within it: Play Mode and Edit Mode. In the Edit Mode of Unity, objects in the Scene View, as seen in **Figure 6**, are modifiable. Objects can be moved and rotated, and other properties of objects can be changed as well. The results of your changes can be seen in both the Scene View (**Figure 6**) and the Game View, which is seen in **Figure 7**, and these changes are the starting point for the scene in the Play Mode. The Play Mode starts when the ”Play” button is pressed, which is located at the top in the middle of the Unity Editor. This button can be seen in **Figure 7** as well. Unlike the Edit Mode, changes that occur to the scene during the Play Mode do not persist after exiting the Play Mode. This allows the user to test scenarios in the Play Mode without changing the starting scene and allows for rapid iterations of edits and testing. On the other hand, when tools are provided for the play mode for scene manipulation, such changes to the scene must be selectively saved and this persistency implementation is additional work for play-mode plugin developers.

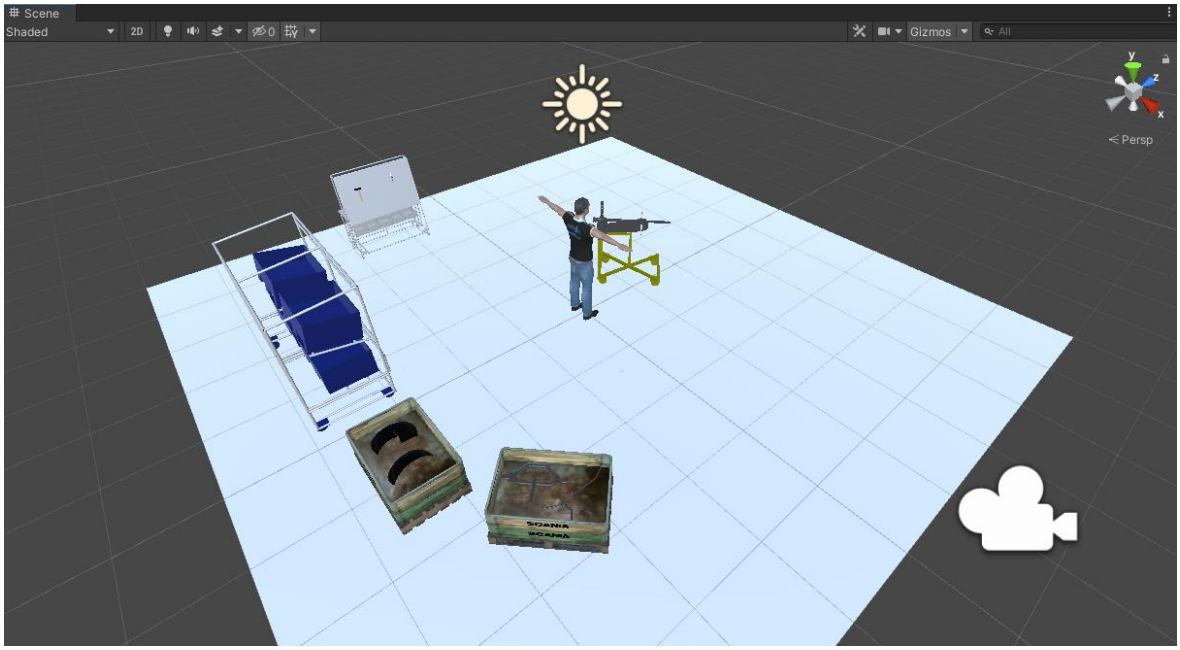


Figure 6: Scene View within the Unity Editor

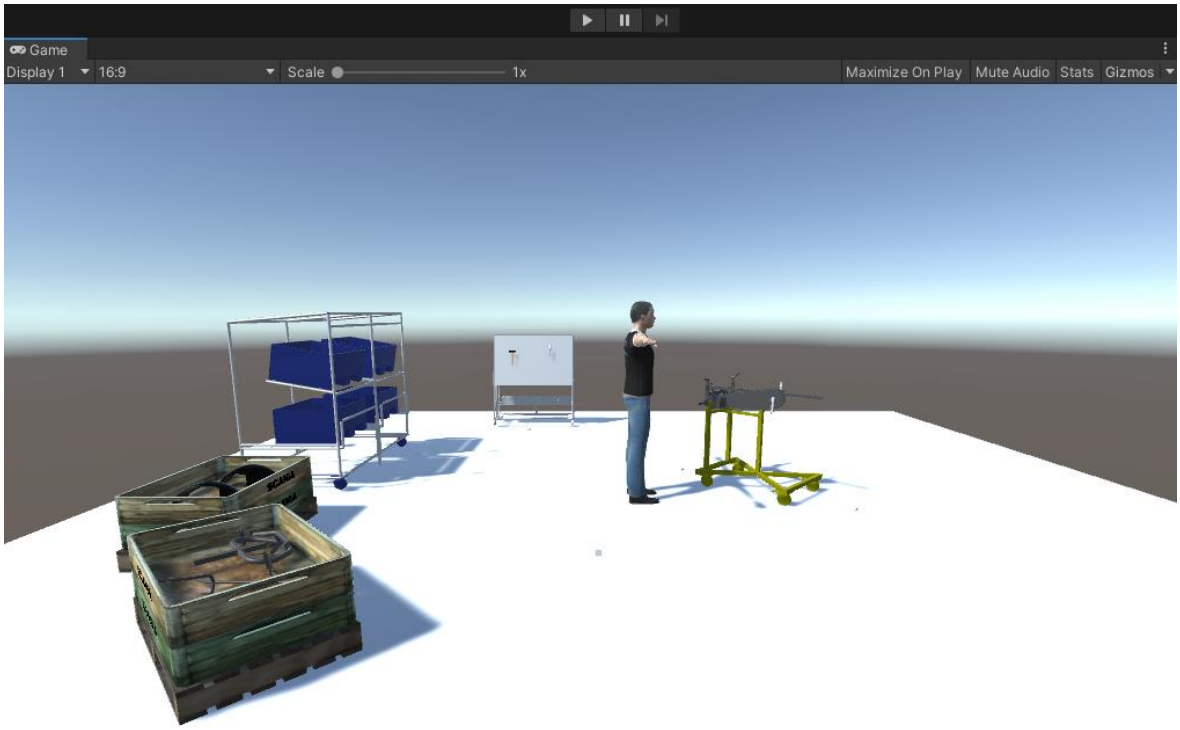


Figure 7: Game View within the Unity Editor

3 VIEW ORIENTATION OPTIMIZATION METHODS

The first objective of this research is to investigate how the “best view” of an object can be determined and implement this into MOSIM by using Unity. As mentioned by Wang, viewpoint selection methods are still progressing but that there is no unique solution that will work for any type of object, as the acceptability of the results depends on the application and the methods used (Wang, 2011, p. 556). This is especially true when determining the “best view” as it can be quite subjective since it depends on the viewer. Approaches applicable for this problem are presented in the following subsections.

3.1 Machine Learning

Machine Learning (ML) is typically seen as a subset of AI in which an algorithm is improved through iterations of testing and experience. By providing training datasets to the system, these iterations change values within the algorithm and would eventually result in a system that is capable of successfully predicting the result a percentage of the time. This percentage is usually dependent on the size of the dataset provided, the number of batches, and the number of epochs used during the learning process. Batches are the number of samples from the dataset that are analyzed before updating the values in the algorithm while epochs are the number of times the entire training dataset is analyzed. (Mitchell, 1997)

ML could be applied to this research by providing the models of the objects to the software and letting the ML software determine the algorithm for the best view. Implementation of ML software could be quite difficult for this application as datasets need to be provided which already have either scores for the views or a classification as an accepted best view or rejected best view attached to that particular view. As mentioned, larger datasets typically train the system better so many views would need to be provided. In addition, Unity does not natively support ML, so some add-ons or workarounds would be needed in order to fully implement ML into this application.

3.2 Weighted Metrics

As mentioned, a score could be attached to each image for ML. By using varying metrics measured within Unity to calculate a score for each view, it is possible to attach these scores to different views of the objects. A simplification of the ML learning process, which is possible to natively implement in Unity, is the utilization of weighted metrics to calculate the score of the view. These weights can be optimized, similar to the algorithm that is typically used in ML, and this optimization can be considered a simple form of ML.

In Unity, the first step in this process is to isolate the subject object from the rest of objects in the scene to prevent other objects from being analyzed. Because of limitations within Unity, the whole 3D object would be difficult, if not impossible, and time consuming to analyze. Instead, the object will be rotated to different orientations depending on the user's specifications. Varying properties of the object can be obtained from Unity and used for each view to determine different metrics. Some of the properties include the mesh, bounding box, and center of mass. The mesh is Unity's main graphics primitive, which are basic elements typically consisting of lines, curves, and polygons (Unity Technologies, 2020). The mesh is comprised of vertices which create edges, and these edges define the faces of an object, even if the object is complex (Michigan Technological University, 2010). These can be seen in **Figure 8**. An example of this can be seen using the ratchet in **Figure 9**, where the black lines on the ratchet are the edges of the triangles. The bounding box is essentially the smallest box whose size is the dimensional limits, in reference to the world axes, of the object. An example of this is also shown in **Figure 9** using the green lines.

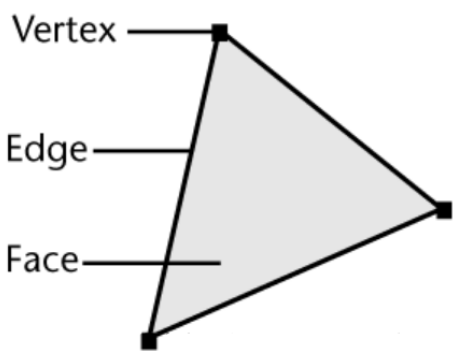


Figure 8: Visualization of the components of a Mesh (Modified: Autodesk Help, 2016)

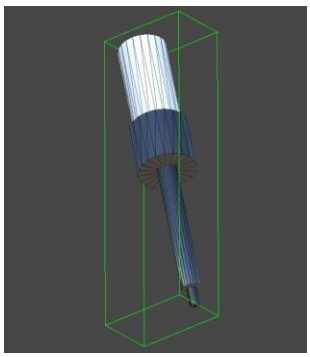


Figure 9: Visualization of a Bounding Box using the Ratchet

Each metric has its own method of being determined using certain properties of the object. It is then be possible to use a combination of the results of the metrics of each view to calculate the "best view." This "best view" optimization procedure will be discussed later in Section 3.3. An explanation of each metric, along with how it is determined, can be seen in the sections below.

Projected Area

The projected area can be described as the area of an object's shadow when a light is cast upon it. This idea can be equated to how large an object is in the camera's view. Typically, the more area the object takes up in the camera's view, the more detail the view is showing. This idea can be seen in **Figure 10** and **Figure 11**, where **Figure 10** has a view with less projected area than **Figure 11**, and consequently appears less detailed as well.

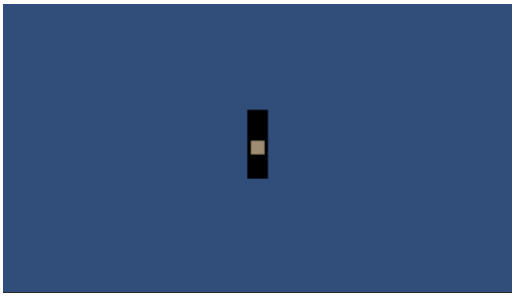


Figure 10: View of Hammer with *less* Projected Area

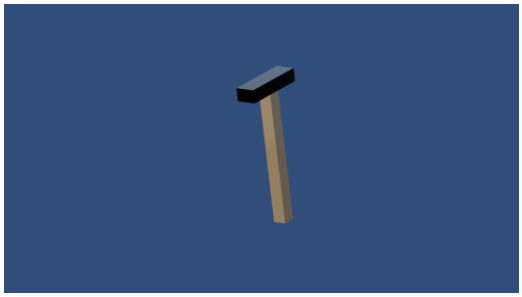


Figure 11: View of Hammer with *more* Projected Area

To implement this into Unity in a measurable way, the implementation consists of using the image shown in the game view and saving the pixels from the game view on the user's screen to a "2D Texture". This texture allows Unity to access each pixel from the texture without the need of rereading the pixels from your screen each time it needs to access a pixel value. Instead of analyzing the pixels in red-green-blue-alpha values (RGBA), a function of the Texture class was used which converts the RGBA values to a single grayscale value and allowed for a simpler analysis. The size of this texture depends on the resolution of the user's monitor. To save time while processing all the pixels, a user-defined skipping constant, "Pixel Skip Size," was added that allows the user to skip over some pixels in the texture.

This means that it is possible to only analyze every second, third, or fourth pixel (or more if needed) instead of every pixel.

The background color in the game view can be set to a specific color by changing a property of the camera. The value for this color is saved and is used when scanning through the pixels in the game view. Taking into account the user-defined skipping constant, it is possible to compare the color of the pixels to the saved background color value. If the pixel does not match the background color, it is assumed to be part of the object. The ratio of pixels of the object to total number of pixels analyzed results in an approximate percentage of the screen filled by that view of the part. It is considered approximate as the user may choose to skip over some pixels which means that the part may actually fill more or less of the screen but after some testing, this percentage is usually not significantly different from the true value. Repeating this for each view provides the view that fills the screen the most. This process can be visualized as seen in the flowchart in **Figure 12**. For each view, the percentage of the screen that the object fills is saved into a Static List. A Static List is a list that will persist and can be referenced, even if no instance of the script (i.e. code), in which the list is created, exists in the given scene. This means that these values will not be deleted when switching to a new view or new object. Static lists will be created for each metric and are then used later when doing the normalization and weighing of the scores and final selection of the best view.

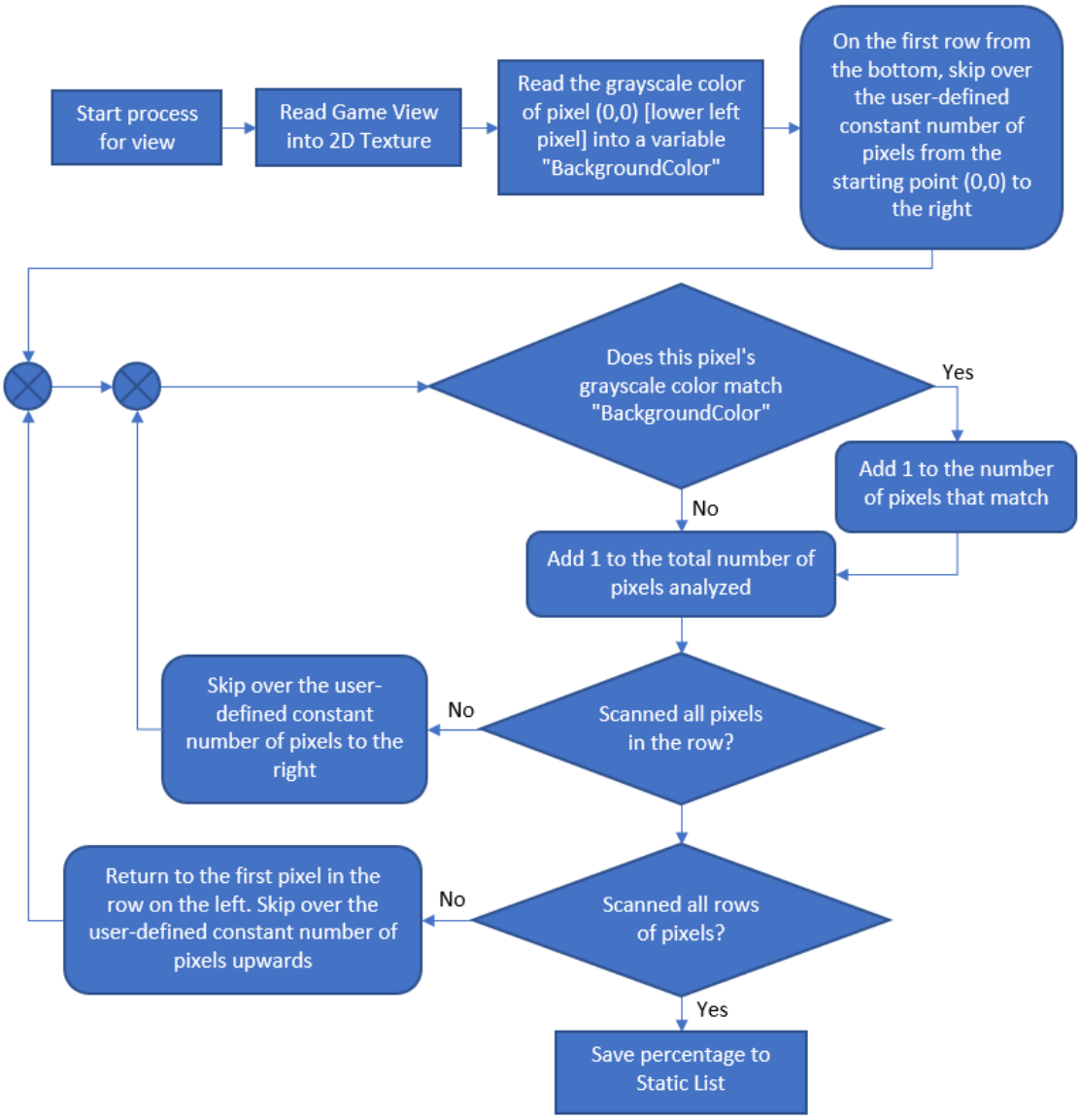


Figure 12: Logic Flowchart of the Projected Area

Ratio of Surface Area of Visible Sides to All Sides

The mesh of the object is implemented to analyze the surface area of the object. By using the vertices of the mesh, it is possible to find which faces of the mesh are visible from a specific view (i.e. from the location of the camera). At the same time, it is also possible to determine the total surface area of the object from these vertices. By having a larger ratio of visible surface area to total surface area, the view is typically more detailed.

In Unity, there is a feature called Raycast. This feature simply creates a ray, or line, between two points and checks to see if this line intersects a collider, which, for this metric, is simply

the mesh of the object. Specifically, the ray is cast from the camera's location towards each of the vertices of a triangle of the mesh. If all three points of the triangle can be hit without the ray intersecting the object, that face is visible, and the area of that triangle is added to the visible surface area total. Regardless if the mesh is intersected by the ray, the area of the triangle will be added to the total surface area of the mesh. Two examples of these triangles can be seen in **Figure 13** and **Figure 14**. Rays (red, blue, and green lines) were extended from the camera (out of view in the image) to the target points of a triangle. A triangle visible to the camera is shown in **Figure 13** and the rays hit the vertices but do not intersect the mesh, while a different triangle that is not visible from the camera's perspective is shown in **Figure 14** where the rays intersect the front side of the mesh in order to reach the target vertices. This is noticeable as the rays are not approaching vertices on the visible side of the ratchet in **Figure 14**, but instead going through a face of the mesh.

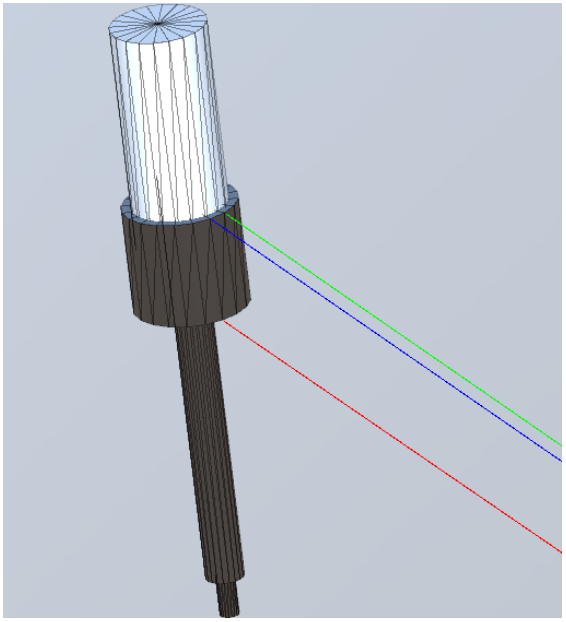


Figure 13: No intersection with the Object's Mesh

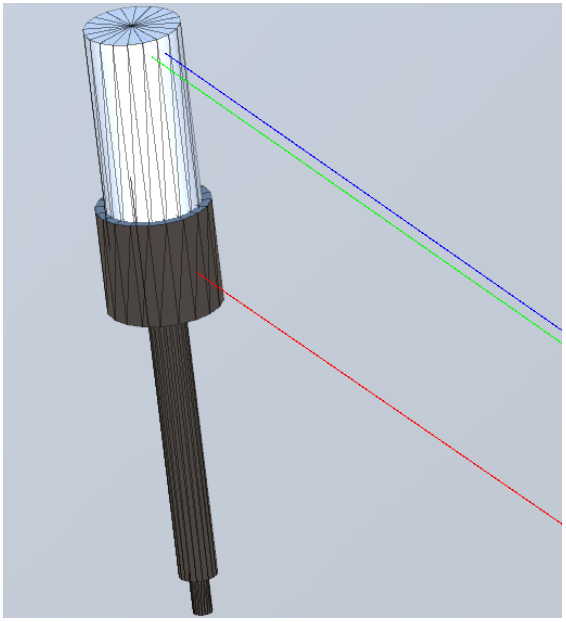


Figure 14: Intersection with the Object's Mesh

In the implementation, a check is completed to see if the ray intersects the mesh on the way to that vertex, but if the ray were to actually go all the way to the vertex, it still would be considered an intersected point, even if the ray doesn't pass through the mesh in another location. Therefore, the ray is scaled slightly to 99.99% of the distance between the camera's location and the vertices of the triangles. This way, it will not intersect any of the vertices,

but instead, approach the vertex. This way, it will only intersect the mesh on the way to hidden vertices. The logic for the checking process and surface area totals can be seen in **Figure 15**. Once all the triangles have been analyzed for that view, the visible surface area ratio is saved to a Static List, like the Projected Area process.

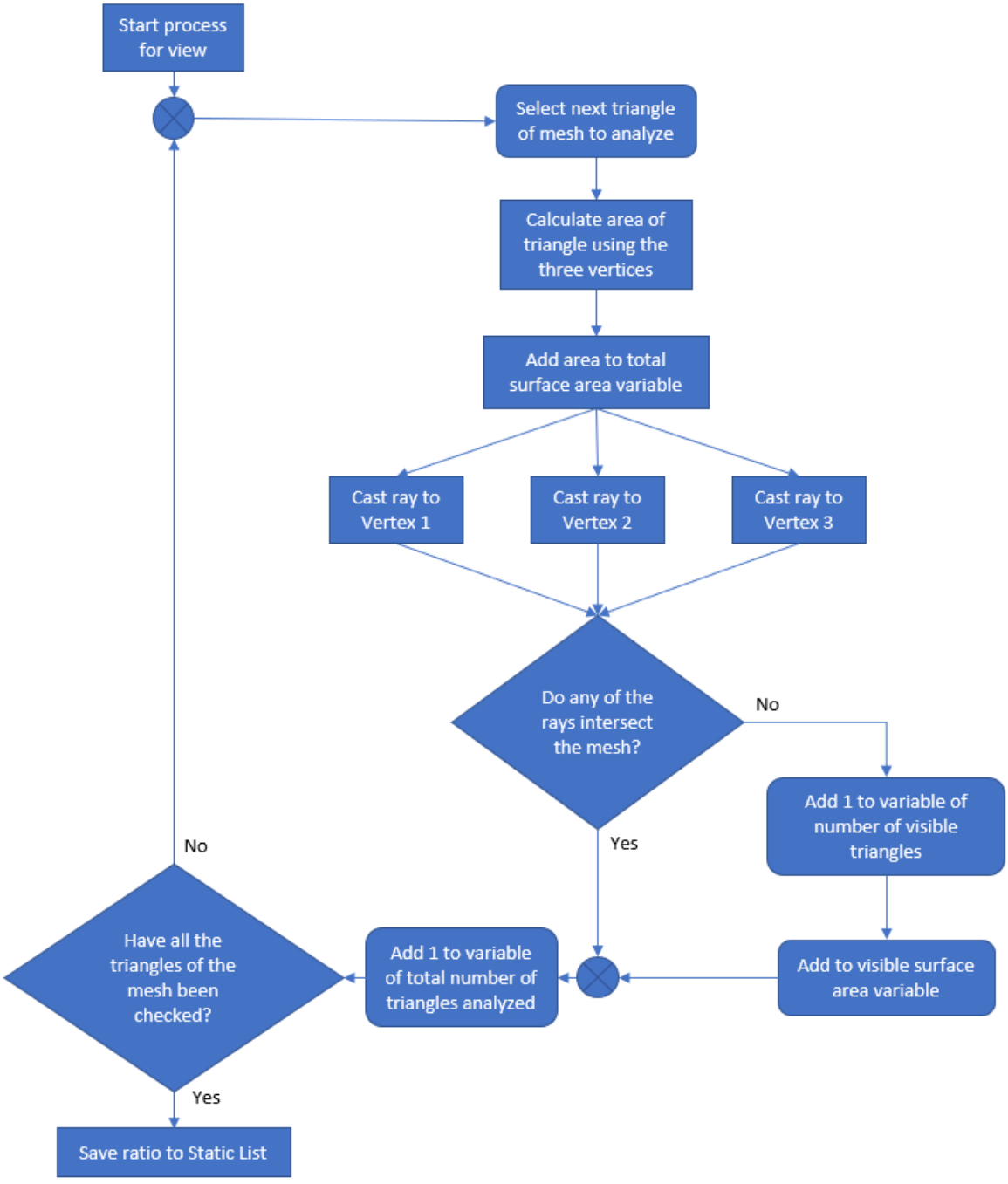


Figure 15: Logic Flowchart of the Raycast Progress

Number of Visible Triangles to Total Triangles in the Mesh

This metric is related to the ratio of the visible surface area, but instead of a ratio showing how much of the surface of the object is showing, this metric indicates how detailed the view is by the number of visible triangles of the mesh. Smaller triangles are needed for the mesh when there are curved features or more detailed features on the objects. When counting the number of the visible triangles, the view with the most triangles will be considered the best view for this metric as there tend to be more triangles when there are small triangles. **Figure 16** is an example of how there are fewer triangles for a mesh when the object is not very detailed, while **Figure 17** shows an example of a more complicated mesh requiring many triangles to describe the shape of the object.

Figure 16: Hammer with only a few large mesh triangles

Figure 17: Screw with many small mesh triangles

This process uses the same flowchart as seen in **Figure 15**, but uses the number of visible triangles and the total number of triangles analyzed for its calculation. The ratio is calculated, as seen in Equation (1):

$$r = \frac{v}{T} \tag{1}$$

where v is the number of visible triangles in the mesh, T is the total number of triangles in the mesh, and r is the ratio of visible triangles to total triangles. This average is then saved to a separate Static List than the ratio of visible surface area.

Center of Mass

The center of mass of the object is important to the view orientation because in most cases, when an object is oriented in a "normal" way, the object usually has a center of mass situated lower in reference to the world's y-axis (up-down) in order to keep the object from tipping over or moving. In this way, having a lower center of gravity can be preferred when finding the "best view" of the object. At the same time, after some testing, it seems that adding in another factor with the center of mass helps orient some objects better. By adding the world's x-axis (left-right) reference for the center of mass, when the center of mass is closer to zero (the center of the screen) a better view is typical because, once again, the object is more likely to be in a stable and natural position when the center of mass is located at zero along the world's x-axis. A unique example of center of mass positions can be seen in **Figure 18** and **Figure 19**. **Figure 18** shows the Fixture correctly oriented as compared to **Figure 19**. The up-down center of mass of **Figure 18** (shown by the red dot) is closer to the bottom of the bounding box than the center of mass in **Figure 19**, but both are fairly close to the center of the bounding box in the left-right direction.

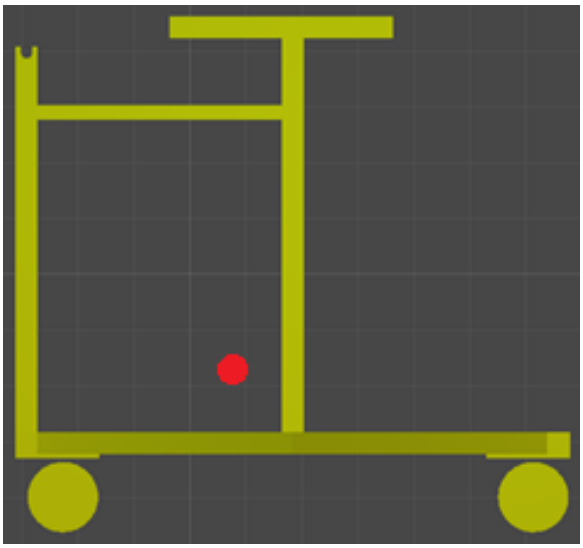


Figure 18: Preferred Orientation using this Center of Mass Method

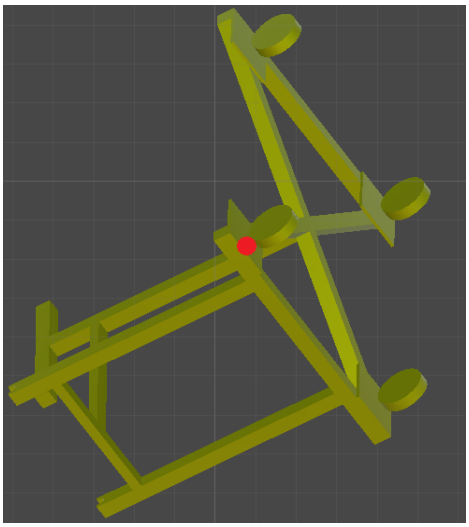


Figure 19: Less Preferred Orientation using this Center of Mass Method

To find the location of the center mass in a particular view, a RigidBody component must be attached to the object. If one is not attached, one is automatically added during the analysis. In this RigidBody, a property can be accessed to find the location of the center of

mass of the object in reference to the world axes. In order to make the value absolute and comparable not only within the same object but with other object's center of mass, each of the center of mass locations are converted to a pixel location in the game view using a function on the camera. With this value, a percentage of the screen can be calculated by dividing the center of mass location distance by the pixel height of the game view for the y-axis center of mass and by half the pixel width of the game view for the x-axis center of mass. These percentages are then saved into two separate Static Lists, as seen in **Figure 20**.

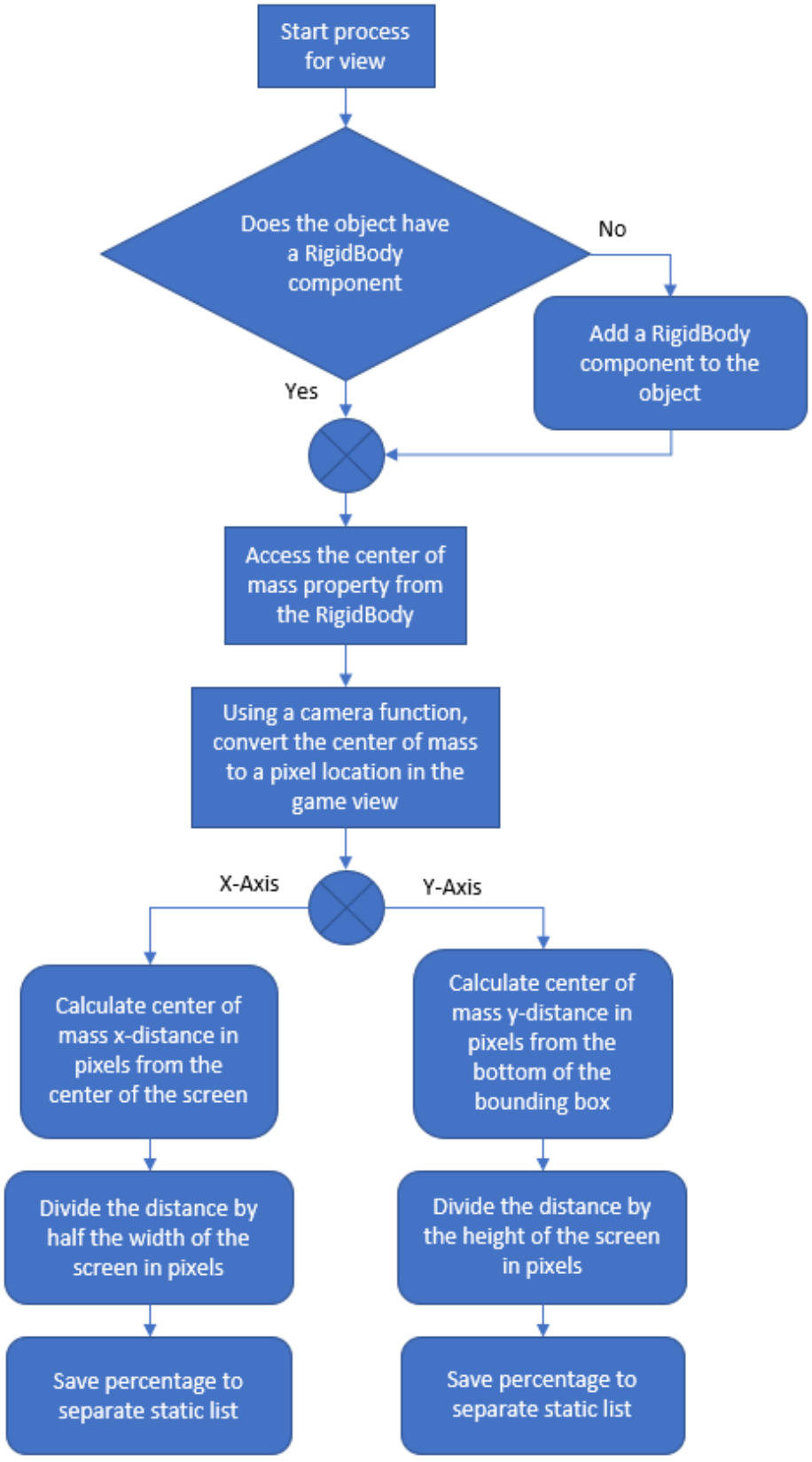


Figure 20: Logic Flowchart of Center of Mass

Visible Edges

When viewing a 3D object, the object's edges can be considered crucial information about how detailed a view is. These edges help the viewer determine how the object's faces are oriented. In Unity, there is an image effect, "Edge Detection," that allows the edges of certain objects to be highlighted by thickening the edge's color or by changing the color. This is accomplished by enabling a new set of cameras that process two different views and create a new view for the viewer. Therefore, before analyzing the part for the visible edges, the original photo camera view (**Figure 21**) is disabled and the edge detection camera view (**Figure 22**) is enabled.

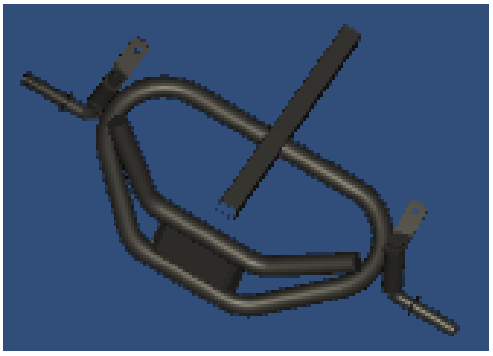


Figure 21: Photo Camera Enabled (No Edge Detection)

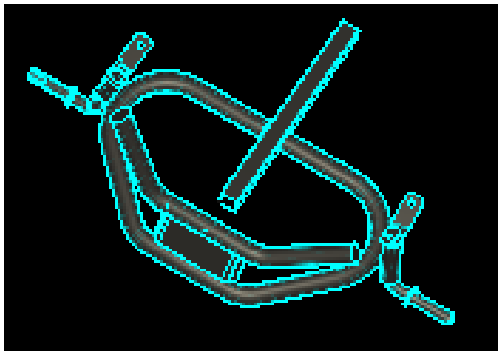


Figure 22: Edge Detection Camera Enabled

In this metric, the edge detection changes the color of the background to make the edges more contrasting. As seen in **Figure 22**, the object's edges are replaced with cyan pixels and the background has been changed to black. By doing so, the view can be analyzed in the same way as the projected area (i.e. scanning through the pixels). In this case, instead of comparing the pixels to the background color, the pixels are compared to the new edge color (cyan) to check if it is an edge. Finally, the ratio for this metric is determined by taking the number of edge pixels and dividing it by the number of pixels analyzed, and then saving this ratio into a static list.

Symmetry

The symmetry in the view can play a factor in the determination of the best view as typically, when the view is symmetric, it means that the view does not describe multiple features of the object. As seen in **Figure 23**, when the object is considered "less" symmetric, the view is displaying more detail, as compared to **Figure 24** and **Figure 25** where the views contain

symmetry but do not show as much detail of the part. Symmetry is usually defined as either a “true” or “false” property, but in this metric, it is measured as a percentage. Therefore, the “best view” for symmetry would have a value of zero or close to zero. This process starts with the direction of symmetry, since a view can be symmetric in the horizontal direction (**Figure 24**), vertical direction (**Figure 25**), or both directions.

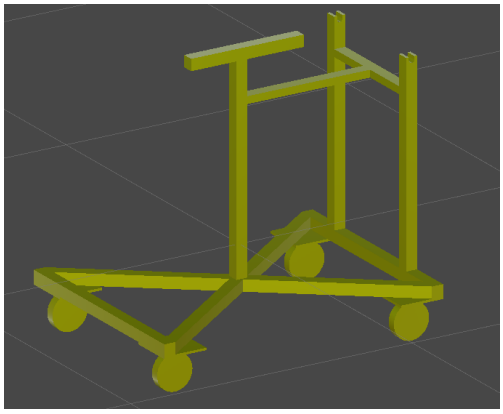


Figure 23: Non-Symmetric View of Fixture

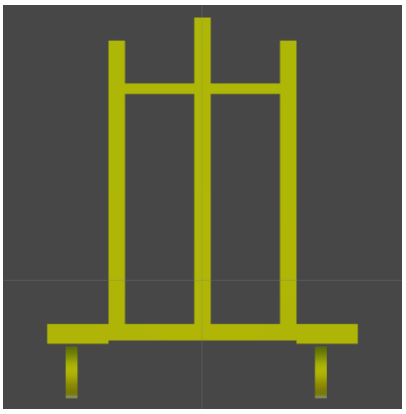


Figure 24: Horizontal Symmetry

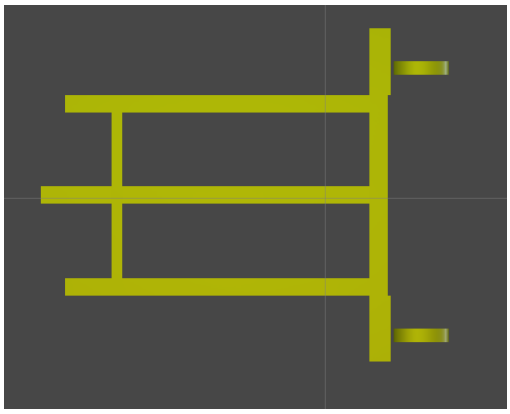


Figure 25: Vertical Symmetry

The same texture used in the Projected Area metric is used for the symmetry. For analyzing symmetry in the horizontal direction (**Figure 24**), the pixels on the left side are compared to the opposite pixels on the right side. This is completed for each row (accounting for the user-defined skipping constant again). Similarly, for the vertical direction (**Figure 25**), pixels at the bottom are compared to the opposite pixels on the top side. This is completed for each column (accounting for the user-defined skipping constants again). The score for the view is the average between the horizontal percentage and the vertical percentage. A visualization of this process can be seen in **Figure 26** and **Figure 27**.

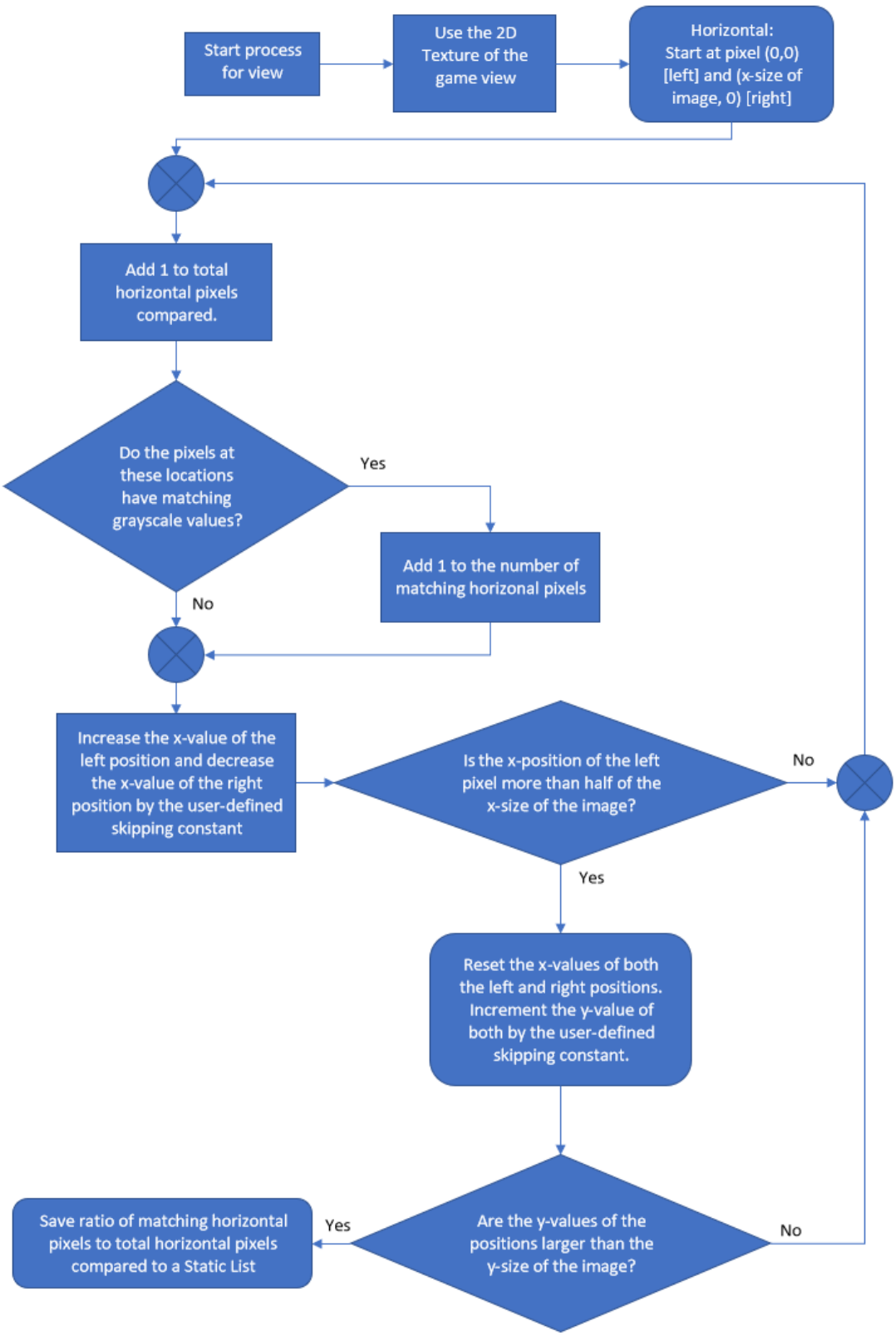


Figure 26: Logic Flowchart of Symmetry (Horizontal)

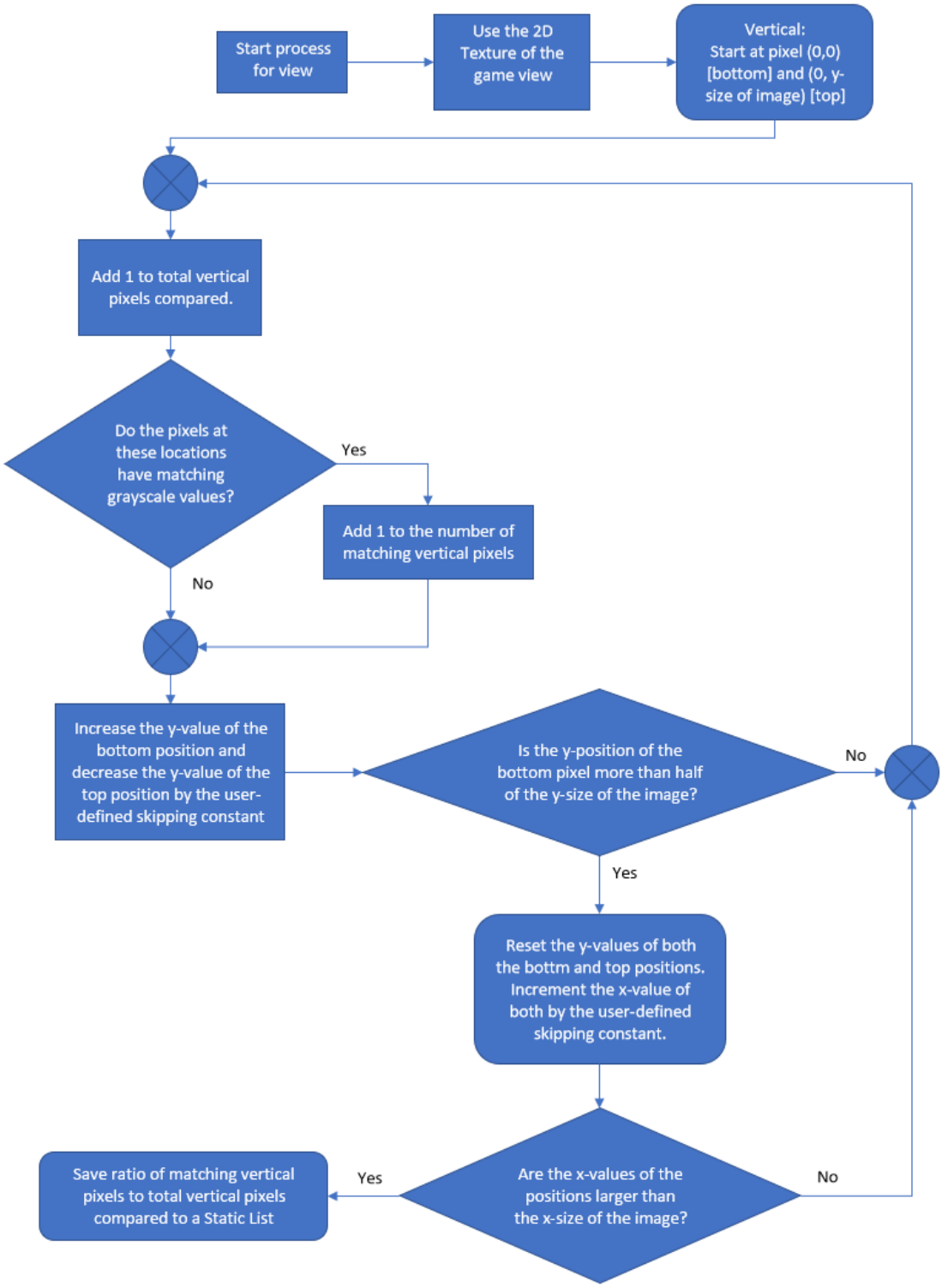


Figure 27: Logic Flowchart of Symmetry (Vertical)

3.3 Optimization Procedure

With the method of calculating each metric defined, these methods can be applied at specified orientations of each object. These orientations are defined by the user in the yaw, pitch, and roll directions. For an object, the yaw (green), pitch (red) and roll (blue) are defined with the circular lines as seen in **Figure 28**.

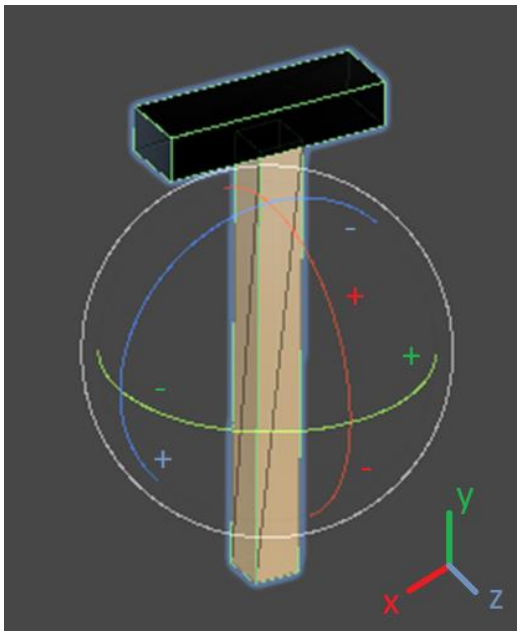
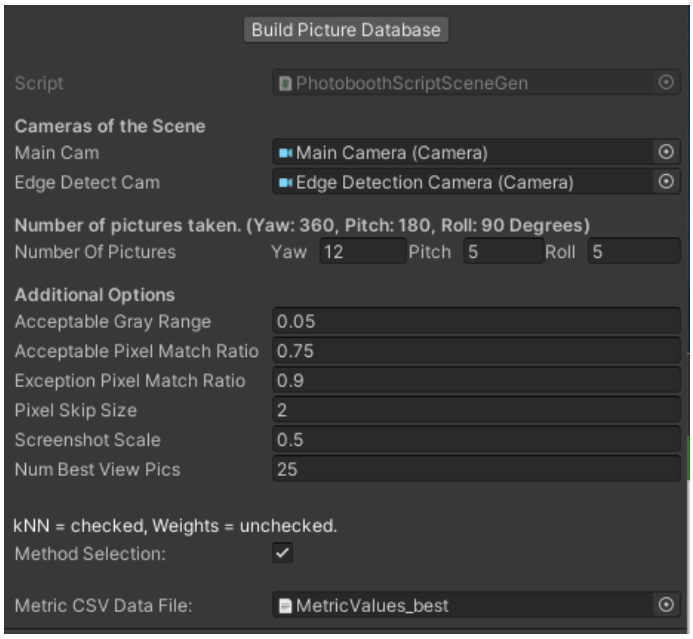


Figure 28: Yaw, Pitch, and Roll Illustration

For this process, the yaw has a full range of 360 degrees, the pitch has a range of 180 degrees, and the roll has a range of 90 degrees. These values will allow any orientation of the object to be obtained. Prior to starting the process, the user should define some parameters including how many photos to take for each axis (yaw, pitch, and roll) and the weights of the metrics (how important they are relative to each other). There are a couple additional parameters that can be defined including the acceptable percentages for symmetry, which is also used for orienting the object, an acceptable grayscale range to allow the colors to be slightly different for symmetry but within a small range, number of "best view" images to be saved (based on their scores) and the scale of those images, and the user-defined skipping constant if the user would like to speed up the process at the expensive of accuracy. These options can be seen in **Figure 29**(a), and short descriptions and default values are provided in the tooltip for each parameter in the implementation in Unity.

(a) Weight Presets Selected



(b) *k*-Nearest Neighbor Algorithm Selected

Figure 29: Modifiable Parameters for the User

By using these parameters, the previously described metrics, and the number of pictures of the object, it is be possible to determine the best view. Prior to starting the procedure, it is also necessary to select which method to use when determining the best view: Weighted Metrics in Figure 29(a) or the *k*-Nearest Neighbors Algorithm (which will be explained later in Section 3.4) in Figure 29(b). This procedure is first started by selecting the "Build Picture Database" button as seen at the top of each image in **Figure 29**. This button is available when in "Edit Mode" and it will start the procedure immediately by disabling unnecessary scripts in the program and by putting the player into "Play Mode." Once done, the first step in the analysis process is to isolate the object to be analyzed. This is done by instantiating a new copy of that object and placing it in a new scene specifically made for this process. This scene will persist throughout the process as the active scene and the original scene will be loaded and unloaded as necessary to copy the objects to this new scene. Lighting and a camera are also created in this new scene.

For each object, once the object is copied to the new scene, the first step of the process is to figure out the size of the object by using its bounding box. Although there is a provided bounding box in Unity, depending on the orientation at which the object was imported, the bounding box is not always accurate. To account for this, a custom bounding box was created by creating a box that enclosed all the vertices of the mesh of the object. This process must be done every time the orientation or position of the object changes. The object is then centered in the scene using the extents of the bounding box, as the bound's defined center point may not always be located in the center of the bounding box. The extents show how far the bounds are from the center of the box. Once centered, the camera is then adjusted in size to compensate for the different sized objects. This adjustment only occurs during the initial setup of the object in the new scene. During the analysis of the object, the camera does not change. To ensure the object is fully in view for any orientation of the object, the vertical size of the orthographic camera and the distance of the camera from the origin is set to the diagonal length of the bounding box. From here the analysis of the object begins.

Check for Axes of Symmetry

The first task is to determine if the object has any axes of symmetry, and, if so, to set the object to them before rotating the object for calculating the metrics. To do this, the object is rotated every degree around the z-axis (the axis that is aligned with the camera's view) for 90 degrees. Every time it is rotated, the object's bounding box is recalculated and the object re-centered. At each degree, the same process as the symmetry metric (from Section 3.2) is done to find the angle with the highest percentage of symmetry. Once done, the object is rotated to that angle (assuming it meets the user-defined acceptable symmetry percentage; if not, it is returned to its original angle), and then this process is repeated with the other two axes by aligning each one, one at a time, with the camera's view. This checks all three axes for symmetry.

Calculations of the Orientation Angles

To calculate the orientation angles for the object, it is first required to use the user-defined number of pictures in each direction for calculating the increments for rotation. In the Yaw direction, the beginning and end angles are equivalent in terms of orientation (i.e. 0 degrees and 360 degrees). In **Figure 30**, an example is presented where 8 positions are requested for Yaw, 5 positions for Pitch, and 3 positions for Roll. In this example, this means that the number of locations for pictures (i.e. the dots in **Figure 30**) for Yaw is equal to the number of segments that 360 degrees will be divided into.

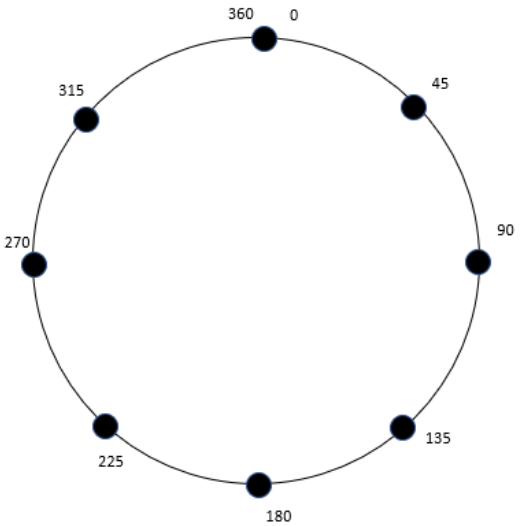


Figure 30: Orientation Angles Visualization

Meanwhile, for the Pitch (0-180 degrees) and Roll (0-90 degrees), since the beginning and end angles are not the same orientation, this means that there is an extra orientation for pictures compared to the number of segments that the 180 degrees (Pitch) and 90 degrees (Roll) will be divided into. The increments for each rotation are shown in Equations (2), (3), and (4):

$$\Delta\psi = \frac{360\ degrees}{p_{yaw}} \tag{2}$$

$$\Delta\theta = \frac{180\ degrees}{p_{pitch} - 1} \tag{3}$$

$$\Delta\varphi = \frac{90\ degrees}{p_{roll} - 1} \tag{4}$$

where p is the number of pictures for that rotation direction, $\Delta\psi$ is the increment of the yaw, $\Delta\theta$ is the increment of the pitch, and $\Delta\varphi$ is the increment of the roll.

Once the increments have been determined, a dummy object is created for the sole purpose of rotation the object through each orientation to make a list of the Euler Angles. This method allows for the individual angles to modified by adding the increments for Yaw, Pitch, and Roll. By using a dummy object with no mesh, the time required to complete this is minimized as there is no need to wait for the next frame of the game view to obtain the values of the Euler Angles. The method used to rotate the object and obtain the Euler Angles relies on loops. Yaw, Pitch, and Roll will all start at 0 degrees. Yaw is incremented through each angle until it reaches 360 degrees. At this time, the Pitch is incremented, and Yaw is reset to 0 and then proceeds to be incremented again. This is repeated until the Pitch reaches 180 degrees. At this time, the Pitch and Yaw are both reset to 0 degrees and the Roll is incremented. This process is repeated until the Yaw, Pitch, and Roll all reach their max values of 360, 180, and 90 degrees, respectively. Each time a value is incremented, the Euler Angles for the object are saved to a list. This process can be visualized in **Figure 31**.

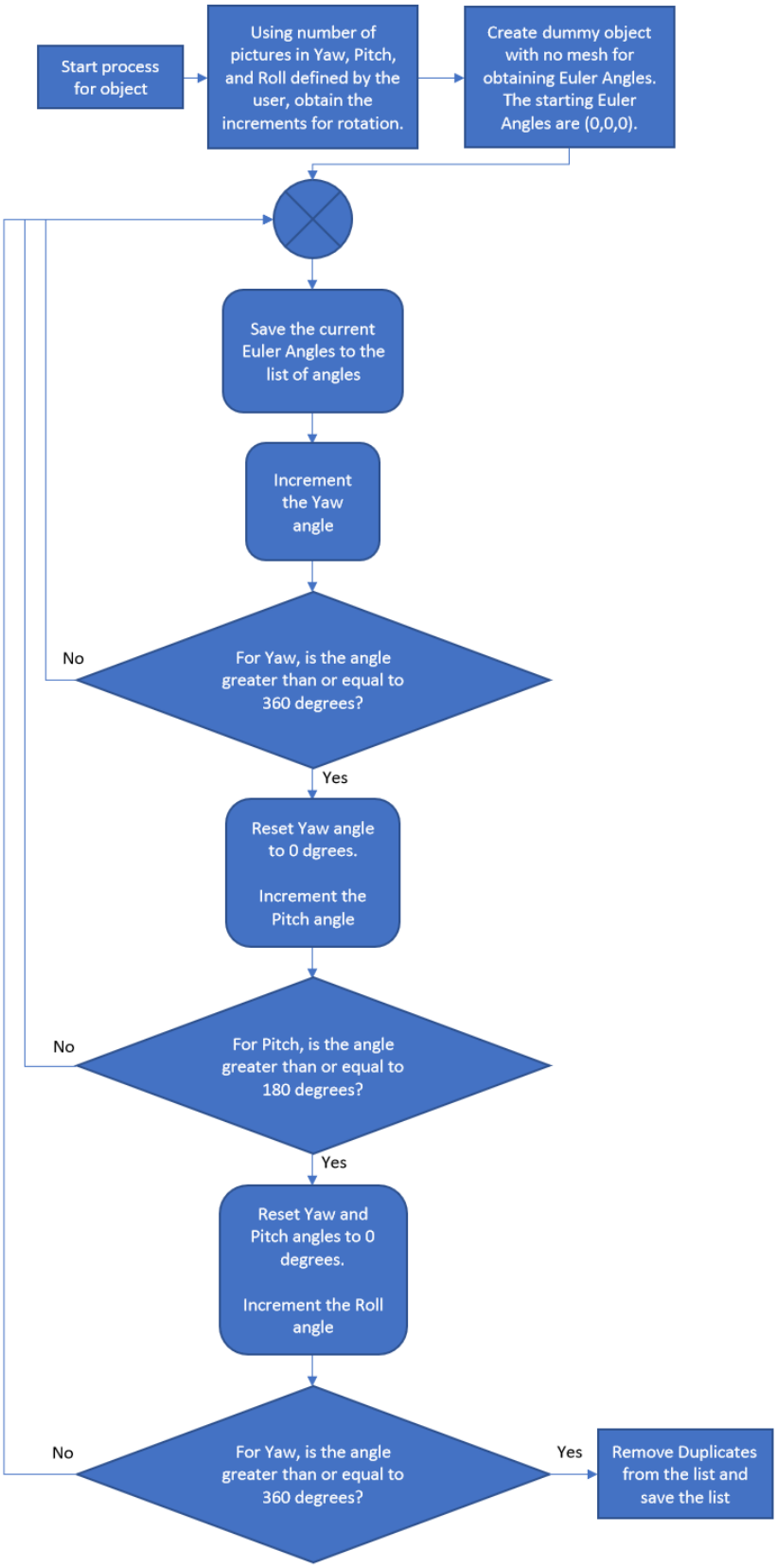


Figure 31: Logic Flowchart of Orientation Angles

Once all the Euler Angles are saved, for efficiency, it is then necessary to scan through the list for any duplicate orientations, as it is possible that some orientations are repeated, depending on the rotation increment for each axis. The duplicates are removed from the list, which leaves only one of that value. After checking for duplicates, the list is finished.

Calculating Metrics

After obtaining all the orientation angles, it is required to rotate the object to be analyzed to each orientation. The basic procedure for this is to rotate the object to the next orientation in the list of Euler Angles, finding the new bounding box for this orientation, and re-centering the object in the camera's view without changing the camera's settings (as described in Section 3.3). Finally, all the calculations, as described in Section 2.5, are completed.

Normalization of Metrics

To use all the different values gained through the variety of metrics, the metrics must first be scaled to the same range of values. To do this, normalization is used to bring the numbers between zero and one. This process is generally completed in such a way that the maximum and minimum are found for the list of one of the metrics and then Equation (5) is applied to each value of that list. This process is repeated for each list.

$$x_{i_{norm}} = \frac{x_i - \min(x)}{\max(x) - \min(x)} \tag{5}$$

In Equation (5), x is the list of values for that metric, x_i is an individual value that is being normalized, and $x_{i_{norm}}$ is the new normalized value. The results are between zero and one, where one is considered the best and zero is considered the worst. There are a few cases that vary, which are corrected to match the others:

i. Center of Mass (y-axis [up-down]) – currently, lower value is better: Because the values were recorded using the distance to the center of mass from the bottom of the bounding box and that the center of mass cannot be outside of the bounding box, all of the values will be positive. Therefore, using Equation (5) will be necessary, but afterwards, since zero is currently the best value and one is the worst, it is necessary to subtract one from every value in the list and take the absolute value. This results in a reversed order of the list, making the values that used to be close to zero now be close to one and vice versa. The resultant list is now in the proper order with one being the best and zero being the worst. See **Figure 32** for the visualization.

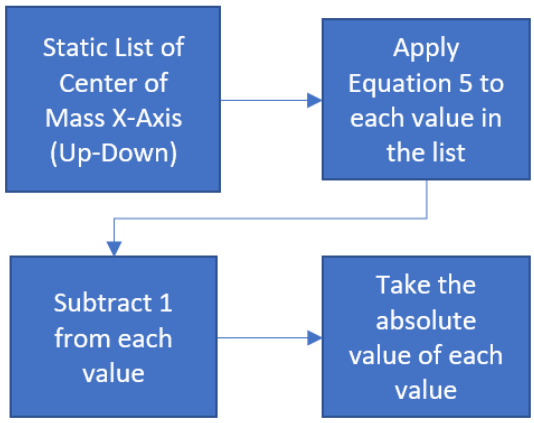


Figure 32: Logic Flowchart for Normalization of the Center of Mass (up-down)

ii. Center of Mass (x-axis [left-right]) – currently, negative numbers are possible and values close to zero are better: Prior to using Equation (5), it is necessary to take the absolute value of all the numbers in that static list to have only positive numbers. This is performed because whether the value is negative or positive does not matter, but only the distance from zero matters. Now Equation (5) is applied to every number in the list. At this point, closer to zero is better but it is necessary to reverse it so closer to one is better. Therefore, subtracting one from every value in the list and taking the absolute value is required. This results in a reversed order of the list, making the values that used to be close to zero now be close to one and vice versa. See **Figure 33** for the visualization.

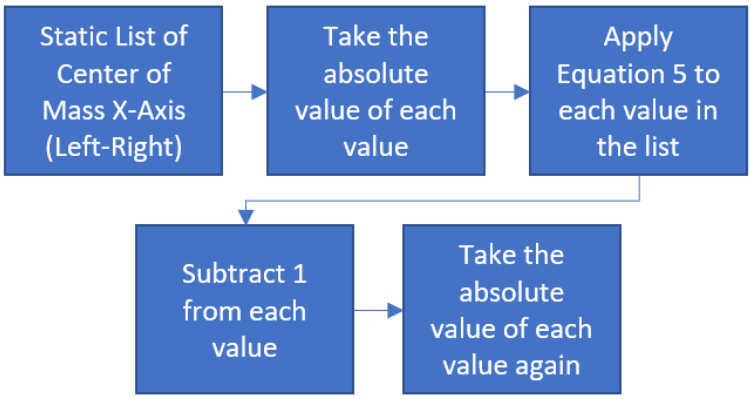


Figure 33: Logic Flowchart for Normalization of the Center of Mass (left-right)

iii. Symmetry – currently, lower value is better: Similar to Center of Mass (y-axis [up-down]), Equation (5) is implemented first. Because less symmetry in the picture typically means that more information is showing, it is then necessary to subtract one from every value and then take the absolute value, as this will reverse the order of the numbers, making the values that used to be close to zero now be close to one and vice versa. See **Figure 34** for the visualization.

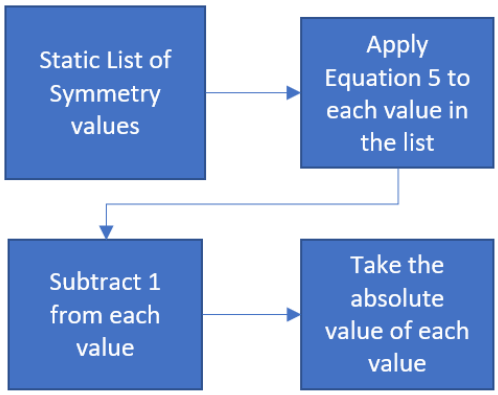


Figure 34: Logic Flowchart for Normalization of the Symmetry

Weighing of Metrics

Now that all the lists have been normalized and are between zero and one, it is necessary to get the total weighted score for each view. In *Table 1*, each column represents one of the object's static lists with the normalized values in them (e.g. Column 1: PV_{V1}, PV_{V2} … Column 2: VSA_{V1}, VSA_{V2}…). Note: Center of Gravity uses the average of the x and y normalized values for each view. Each metric has a user-defined weight (W1…W6, etc.) that will be applied to those scores, which is also shown in *Table 1*.

Table 1: Unweighted Values for Metrics of Each Object (Example)

	Projected View Area	Ratio of Visible Surface Area to Total Surface Area	Number of Triangles in Mesh of Visible Surface	Center of Gravity (average of x and y scores)	Visible Edges	Symmetry
Weight (W)	0.2 (W1)	0.15 (W2)	0.15 (W3)	0.15 (W4)	0.15 (W5)	0.2 (W6)
View 1	PV_{V1}	VSA_{V1}				
View 2	PV_{V2}	VSA_{V2}				
View 3	PV_{V3}	VSA_{V3}				
…						

These weights can be selected in Unity in the inspector, as seen in **Figure 29**, along with the option to have preset values. These preset values allow for the user to quickly switch between different settings (the weights and the additional options). It is possible to rename the presets to make them more recognizable. These values can be saved to and loaded from an external file format known as an "Extensible Markup Language" (XML), which is a format that allows the document to be both human-readable and machine-readable. Saving the different presets that the user created to a location on your own computer using an XML format is done with the "Save Presets" button. By doing so, it is possible to send the XML file to other

users for them to use in their own simulation. The “Load Presets” button allows the user to navigate their own computer for the XML file. If a file was previously loaded, this will be displayed in the label “Previously Loaded Parameters” below the Presets list. The “Clear Presets” button is used for deleting all the presets, creating a new default preset, and clearing the loaded file from the label in the inspector. Before starting the procedure for taking pictures, the preset is selected by the user, which changes the weights in the inspector. By multiplying that weight down each column as seen in *Table 2*, the output is weighted scores for each view. The last step is adding across each row to obtain the final weighted score for each view (e.g. T_{V1}, T_{V2}, T_{V3}…). These final weighted scores are then saved into one more static list to be referenced later.

Table 2: Weighted Values for Metrics of Each Object (Example)

	Projected View Area	Ratio of Visible Surface Area to Total Surface Area	Number of Triangles in Mesh of Visible Surface	Center of Gravity (average of x and y scores)	Visible Edges	Symmetry	**Total**
Weighted View 1	$PV_{V1} \times W1$	$VSA_{V1} \times W2$	…				T_{V1}
Weighted View 2	$PV_{V2} \times W1$	$VSA_{V2} \times W2$	…				T_{V2}
Weighted View 3	$PV_{V3} \times W1$	$VSA_{V3} \times W2$	…				T_{V3}
…	…	…	…				

Selecting the "Best View"

The final step to select the best view is to use the static list saved during the normalization and weighing process. This list of the final weighted scores will be searched for the highest score in the list for that object. When that weight is found, it saves the position (index) in that list and then sets the value of that score to zero. This allows for another search for the second highest score. In this way, it is possible to search for the top *Z* views, where *Z* is the number of different "best views" that the user defined.

Taking Pictures/Screenshots

After selecting the best views and saving the indexes as seen in *Selecting the "Best View,"* those indexes are used to rotate the object. Because all the lists that have been created throughout this process have used the same indexing, the indexes saved for the best views directly correspond to the orientation list. Therefore, it is necessary to rotate the object to the corresponding orientation, recalculate the bounding box, and center the object. Once completed, the process of taking the picture requires that the game view is saved to a scaled 2D Texture, whose scaling depends on the user-defined parameter "Screenshot Scale," which is then converted to Bytes, and finally saved to a picture file—in this case, a PNG file.

Clean-Up of the Scene

Once the pictures are saved, the object is then deleted from the scene, any unused resources (e.g. Textures from analyses and saving of images) are unloaded, certain local variables reset to their original values, and the original scene is re-loaded. Because of the static variables keeping track of which item is next to be analyzed, the script will then proceed with the next object.

Flowchart of the "Best View"

A visualization of the full "Best View" process can be found in **Figure 35**. This serves as a summary of the best view orientation procedure explained in Section 3.3.

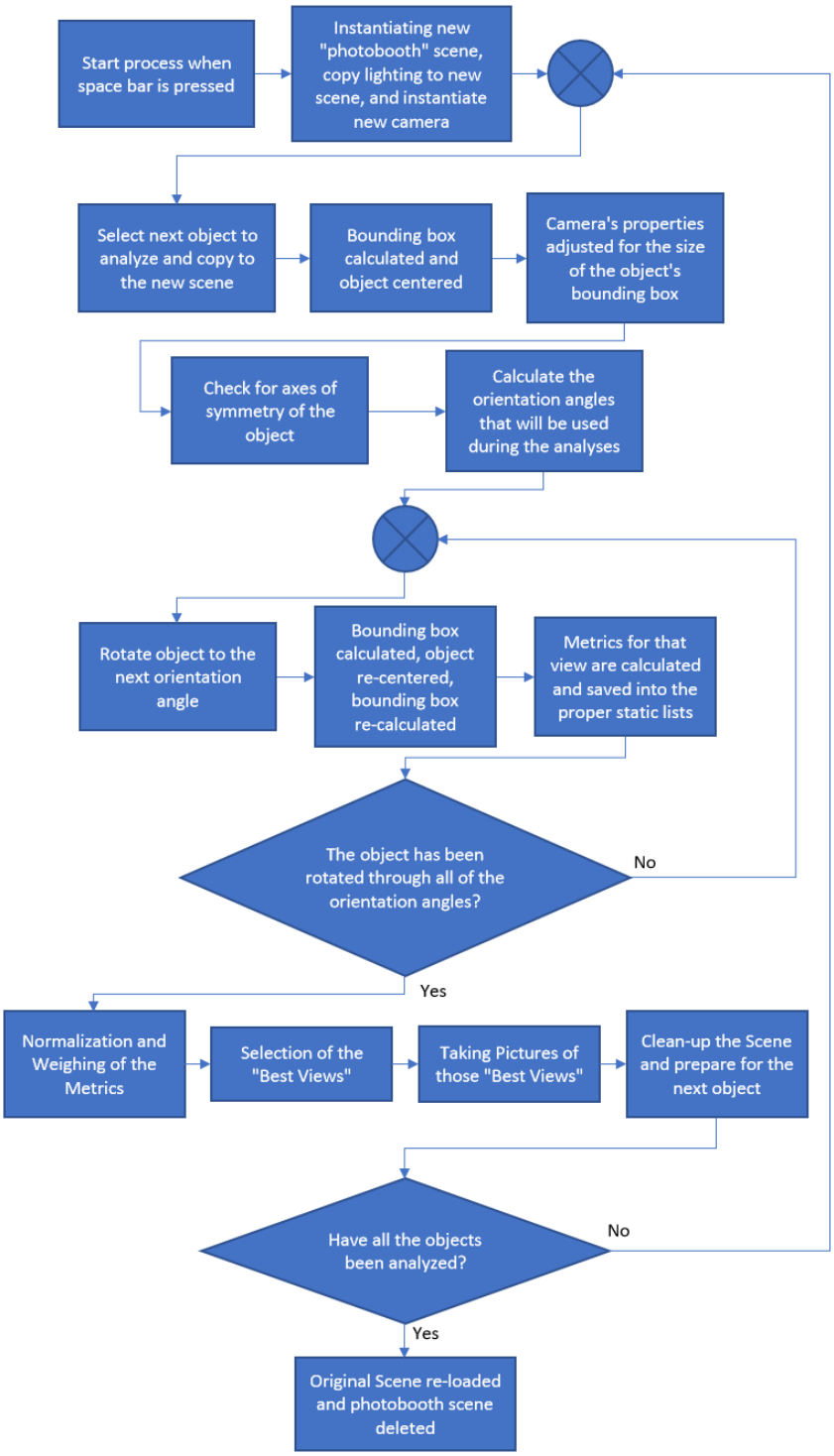


Figure 35: Logic Flowchart of the Best View Orientation Procedure

Optimization of User-determined Parameters

As a final step to this method, optimization of the parameters needs to be done to determine which set of weights would be the best set of default values. With these default values, the user would not have to manually sift through multiple iterations of weights to find the one that works best. This way, the default values would be provided but could be changed if the user deems necessary (i.e. the user considers a parameter to be more important).

Originally, the idea was to use Unity Simulation to run a multitude of iterations of the process using different weight combinations. These weight combinations are found by simply incrementing through values for each weight and adding those values to a list of weights if the weights add up to 1. By doing so, a score could be computed for each weight combination using all the parts analyzed. Unity Simulation could accomplish this quickly and efficiently, but it was later realized that it was not necessary to run the same analysis on each part for each weight combination, but rather run the analysis on the parts once and do computations multiple times using the raw values. With large enough increments for the weights, this would allow a local device to run the simulation rather than needing cloud computing, but with smaller increments, it might still be necessary to run this simulation using Unity Simulation.

The weights were applied to each view for each part. For a part, the best view is selected for that weight and the raw values (i.e. not normalized) were inserted into a matrix of information. This information would be visualized as a cube, where one dimension is the part number, one dimension is the weight combination, and the last dimension is the metric. With this raw information for each part for each metric for each weight combination, the data could then be normalized in a similar manner to the previously explained methods. Instead of normalizing the scores for each metric on a per-part-basis, all the scores for all the parts are normalized relative to each other for each metric. In other words, if each metric is visualized as a plane of data consisting of each row being a different part and each column being a different weight combination, the scores on this plane are normalized in the same manner as previously demonstrated in Equation (5) and the related exceptions. This is done for each plane (i.e. metric). Then for each weight combination, the scores in the metric-part plane could be added up to get a final score for the weigh combination. The highest score

would be the best weight combination as it produced for those objects the highest total score throughout all the objects.

3.4 K-Nearest Neighbors Algorithm

In addition to the weighted metrics approach, another known method is the k-Nearest Neighbors (*k*-NN) Algorithm. This approach is another simplification of ML where the simplicity of implementation and the fact that it is nonparametric and learning-based makes this method appealing (Ni & Nguyen, 2009). *k*-NN uses *k* training points, which are previously determined from the training data, and compares the new data point to these training points. The value of these points can be defined simply as a measured variable. A simple way to implement k-NN is for classification. In **Figure 36**, an example is presented where two variables were measured for different objects and plotted on the x- and y-axes. Each object was already classified prior to plotting, but a new example is added to the plot where the object has not been classified. In this case, *k* can be chosen to include *k* amount of closest data points. The total number of data points for each classification can be tallied and the new point will be labeled as the class with the most data points near it.

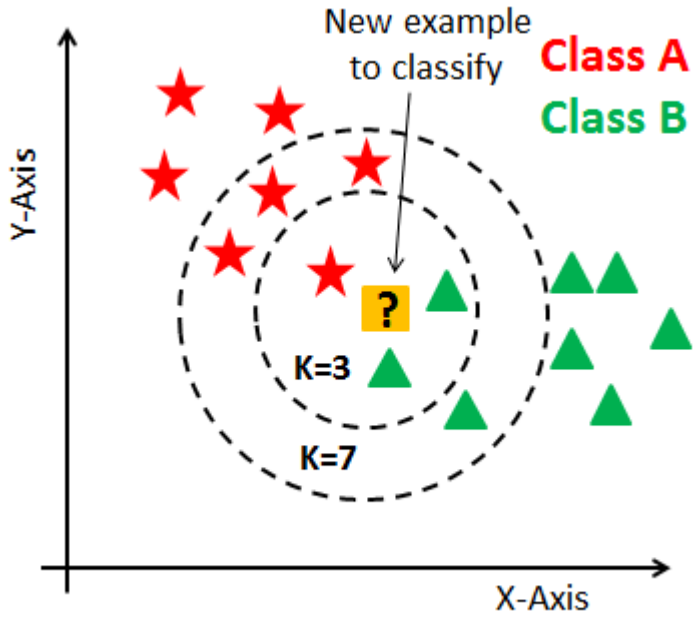


Figure 36: Training Data comparison to New Data Point (Navlani, 2018)

This approach can be used but in a weighted application as well. By taking the distance from *k* number of surrounding points, it is possible to create a total score for the new point. For example, for each of the green triangles, if you take the distances of the three triangles in the "K=7" circle, the distances might amount to 1, 1, and 4. By taking the inverse of each number and adding them together, as seen in Equation (6), a total score is determined for the green triangles class.

$$T_c = \sum_{i=1}^{a} \frac{1}{D_{ij}} \tag{6}$$

where T_c is the total score for class c, D_{ij} is the distance from the new point, j, to the reference point, i, and a is the number of points for that class in the *k*-NN region. The distance from the new point, j, to the reference point, i, can be determined using Equation (7) (Halabisky, n.d.):

$$D_{ij}^2 = \sum_{m=1}^{n} \left(x_{mi} - x_{mj}\right)^2 \tag{7}$$

where, *m* is the current dimension being analyzed, x_{mi} is the value of the reference point for the m^{th}-dimension and x_{mj} is the value of the new point for the m^{th}-dimension. Although it is not possible to visualize a distance in dimensions greater than the 3rd-dimension, by using Equation (7), the distance in n-dimensions can be found and implemented in Equation (6).

To apply this concept to the best view of the objects, it is necessary to first create a list of training data points that can be used. These points are hand-picked, which means that there could be some bias involved but as more data points are added from different users, the accuracy of the results should increase. Example training data points can be seen in *Table 3* for some of the objects in the scene. These points, along with the average values of the best and worst values without outliers removed, are plotted in **Figure 37**. A full list of the training data points for all metrics, along with the respective plots, are shown in Appendix I. Although the worst view values are not actually used in the determination of the best view,

it is useful to have this data to ensure that the best view values and worst view values are distinct enough to decide if a view is of good quality.

Table 3: Example Training Points for the Projected Area

Part	Best View Values	Worst View Values
Mainframe	0.044708	0.024968
	0.050586	0.036614
	0.048373	0.017108
Right-rear Mudguard	0.086318	0.044898
	0.083017	0.084658
	0.0865	0.140309
Right screw	0.102788	0.054859
	0.096645	0.054845
	0.102188	0.070323
Right Steering Link Inside	0.106246	0.06525
	0.106603	0.065267
	0.104702	0.102633
Hammer	0.06847	0.02693
	0.073047	0.02693
	0.070014	0.038325
Ratchet	0.056157	0.013468
	0.054591	0.019009
	0.055891	0.020056

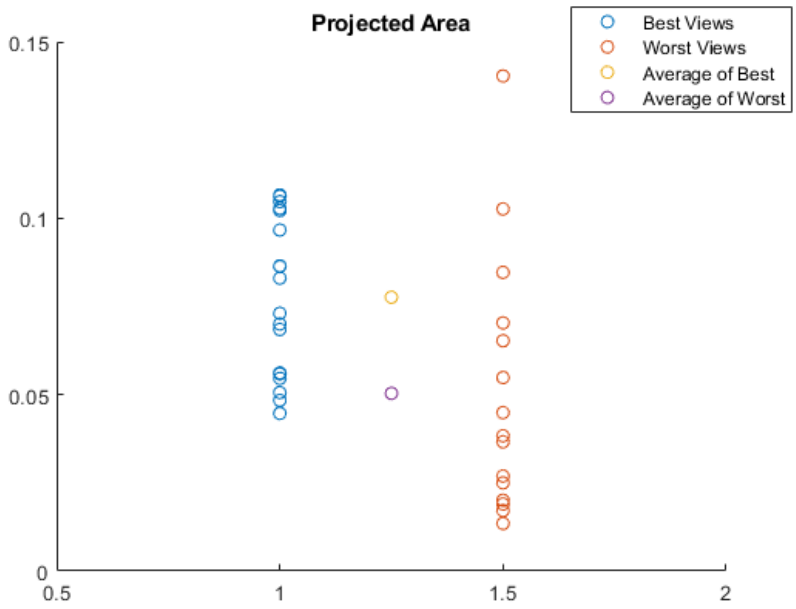


Figure 37: Example Training Points of Projected Area (PA) Plotted

With this training data, the parts in the scene can be analyzed in the same manner as the weighted metrics to obtain their raw values for the metrics. Each part is completed one at a time. For each object, once each view of that object has been analyzed, the raw values are then combined with the values of the best view training data set. With this full list of data, each metric can be normalized in the same way as the weighted metrics method. This allows the raw data of the views, along with the training data points, to be between zero and one and all the metrics become comparable to one another. The normalized value of each metric of the object's view is then used to calculate the distance between that view's data point (which consists of all seven normalized metric values: projected area, visible surface area ratio, center of mass X, center of mass Y, symmetry, mesh triangles, and visible edges) and each of the training data points (which also consists of all seven normalized metric values). This distance is completed as a 7-dimensional distance (see Equation (7)). The distances are then weighed and added as seen in Equation (6). This process is done for each view of the object and the scores are compared. The view with the highest score is considered the best view for that object. This is then repeated for each object, as necessary.

4 SIMULATION CAMERA PATH UTILITY

As previously mentioned, the second objective of this research is to create a system for pre-defining camera locations and orientations during the simulation. These locations and orientations need to be capable of being saved and loaded for future simulations. For now, the system resides in the Inspector in Unity but could always be transitioned into the game view later if needed. The different options of this custom inspector will be described in the next sections and can be seen in **Figure 38**.

Figure 38: Custom Inspector for Camera Path Utility

4.1 Inspector Interface

To have the camera move to a variety of locations with differing orientations, it is required to have a list of timestamps, which are like blocks of information that is used throughout the simulation. The primary information stored is the location and orientation (i.e. rotation) at which the camera needs to be at a specific time. These points are then used to determine an optimal path for the camera to follow in order to be at those specific points at those specific times (this process will be further explained in Section 4.2). When the label above the slider's label "Current Time in Simulation" is "Default camera path enabled," it is possible to scroll the slider through different times, including times between the timestamps, which moves the camera through the optimal path. A second option is to use the "Previous Timestamp" and "Next Timestamp" buttons, which jumps the camera to the proper timestamp relative to the current time in the program.

There are a few different options which allow for "non-default" camera paths, as seen in **Figure 39** and **Figure 40**. If it is needed to move the camera freely, there is an option called "Enable Camera Flying," which allows the user to move the camera around the scene using the WASD keys and to rotate the camera using the arrow keys. The second option below that is "Follow Avatar", which, as the name suggests, enables the camera to follow the avatar during the simulation rather than following the optimal path. Similarly, there is also an option connected to each timestamp, which allows the avatar to be followed during a time interval. The third option positions the camera at a user defined position and orientation while the option is selected. As seen in **Figure 40**, it is possible to add, remove, and rename these static camera locations in order to quickly switch between them. These static camera locations are saved along with the timestamps for the camera path, which will be discussed later. These options override position and rotation that is typically determined by the slider for the time.

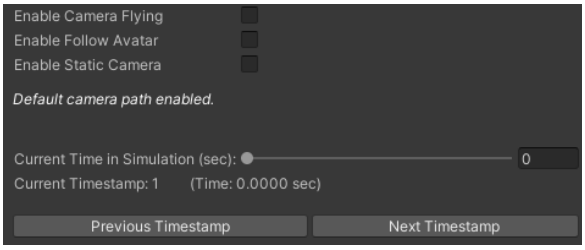


Figure 39: Default Camera Path Enabled

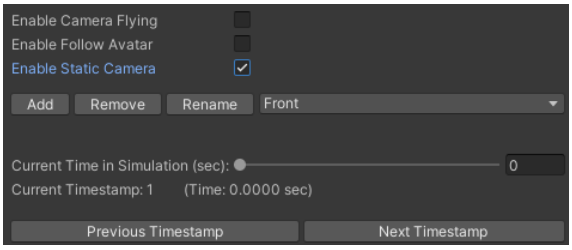


Figure 40: Static Camera Selected (Non-Default Camera Path)

While there are options for non-default camera paths, there are also different options for modifying the optimal (default) path. There are two Timestamp options "Add" and "Delete" and two Timeline options "Save" and "Load," with one miscellaneous option of "Clear All." Using a custom "class" in C#, it is possible to create custom properties for my custom "Timestamps." These properties are simply the location, orientation, time, and follow avatar position/orientation. When starting a new timeline from scratch, the user would move the camera to a specific position while the "Enable Camera Flying" option is enabled. After specifying a time on the slider, the user could press the "Add" button. This takes the time, position/orientation of the camera, and the default follow avatar position/orientation and saves it to a new "block" of information in a list. This list will contain all the saved timestamps. By repeating this process, multiple timestamps will be saved to this list and re-ordered every time a new entry is made to keep the timestamps ordered by the "time" component. There are a few timestamps from the list can be seen in **Figure 38**. Each timestamp can be found under its respective numbered position in the list of timestamps, which have alternating colors for their block of information. The label of the timestamp is red only if the time in the simulation is between that timestamp's time and the next time in the list. There are a couple options for each timestamp as well, but those will be discussed later. Similar to the "Add" button, the "Delete" button can be pressed and an option to delete individual timestamps will be given for each timestamp in place of the original options (circled in green in **Figure 41**). When the "Delete?" button is selected, the timestamp is removed from the list of timestamps and the list is re-ordered to ensure that the list is still in increasing order relative to the time values. Pressing the "Stop Deleting" button, which is circled in blue, will revert the options of the timestamps back to the original options.

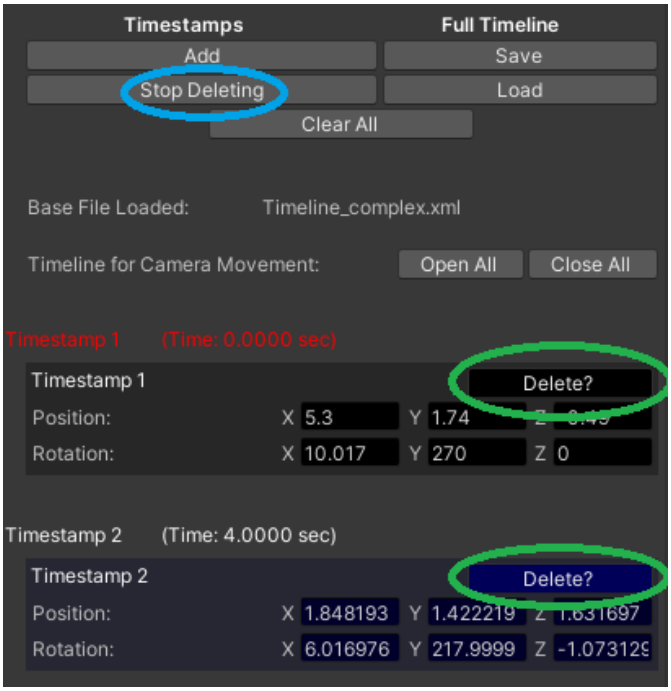


Figure 41: Delete Option for Timestamp

Regarding the Full Timeline options, “Save” and “Load” are used when exporting/importing the timestamps to/from an external file. Similar to the XML file used for the preset weights and additional options, the timestamps can also be saved to a separate XML file that can be shared and loaded later. Saving to an XML file is done by using the “Save” button while loading is done with the “Load” button. If a file was previously loaded, its name will be displayed in the label “Base File Loaded” below the “Clear All” button. This “Clear All” button is used for deleting all the timestamps and clearing the loaded file from the inspector.

Returning to the options for each timestamp, when there are timestamps loaded or added to the list, there are additional options located to the right side of the timestamp block. These options are simple as the “Edit” button allows the user to modify a specific timestamp without manually deleting and remaking that specific timestamp. Selecting “Edit” enables “Camera Flying” automatically so the user can freely move the camera and change the time without modifying other timestamps. It also disables some other options (e.g. Save, Load, Delete, etc.) to avoid conflicts in the editing process. The timestamp is switched to green to indicate that it is being edited and the buttons for every timestamp, except the one being edited, are removed. This is to ensure that the proper timestamp is edited. Two new buttons

appear as well when editing: “Save Current View/Time?” and “Cancel.” All these changes can be seen in **Figure 42**.

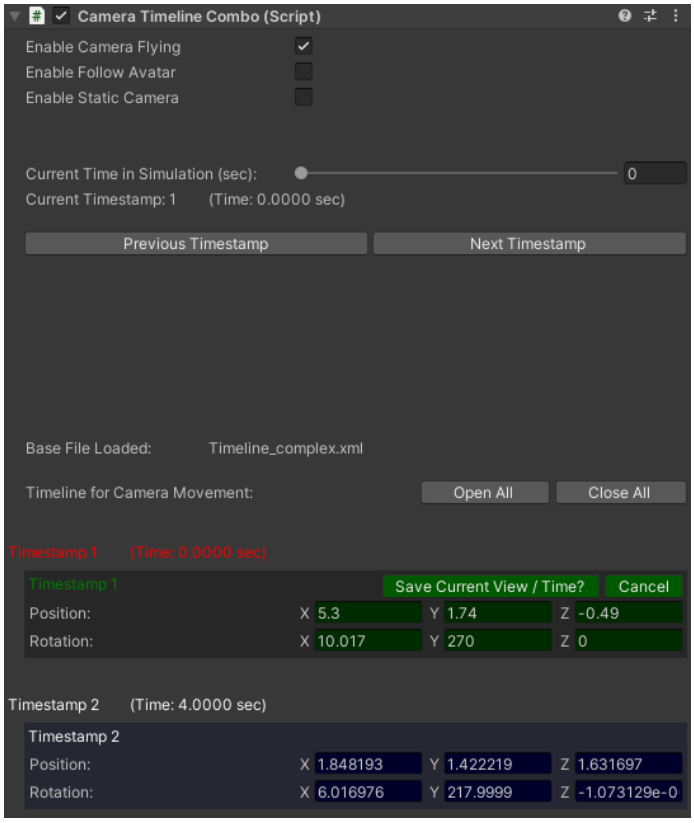


Figure 42: Timestamp Editing

Once you have the new location and orientation of the camera and the new time set, you can select “Save Current View/Time?” and it will automatically delete the old timestamp and add a new one of this new view and time. If you decided that you do not want to edit this timestamp, hitting “Cancel” simply reverts the timestamp’s block to how it used to be without changing the list of timestamps.

The second option on the original timestamp block is “Follow Avatar” which, as previously explained, will follow that avatar during the simulation. As seen in **Figure 43**, Timestamp 1 does not have “Follow Avatar” enabled, but Timestamp 2 does. When enabled, you can “Edit Follow” which allows the user to select the location and orientation relative to the Avatar to follow it. This means the user could follow the avatar from directly in front of it, to the side

of it, etc. When editing, as seen in **Figure 44**, only that timestamp will have options and it will allow you to “Save Follow” or “Cancel,” similar to the editing of the timestamp.

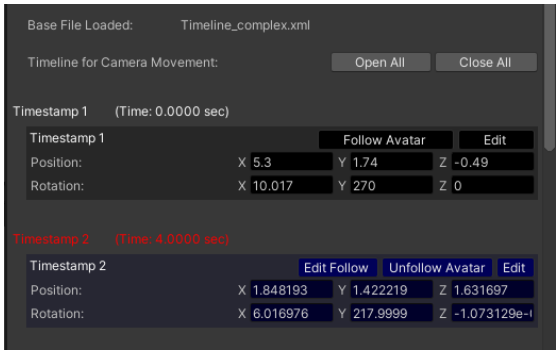


Figure 43: Follow Avatar Options

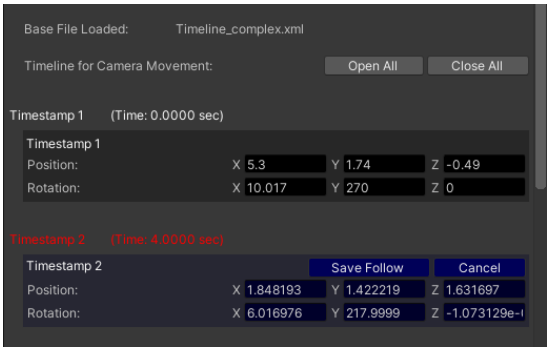


Figure 44: Follow Avatar Editing

For each timestamp, it is also possible to collapse the timestamp so the user can save space while viewing the inspector. The user can close specific ones by clicking on the name of the timestamp (e.g. “Timestamp 1”). There are also the options to “Open All” or “Close All.” As the names imply, “Open All” will un-collapse all the timestamps while “Close All” will collapse all the timestamps. The collapsed timestamps can be seen in **Figure 45**.

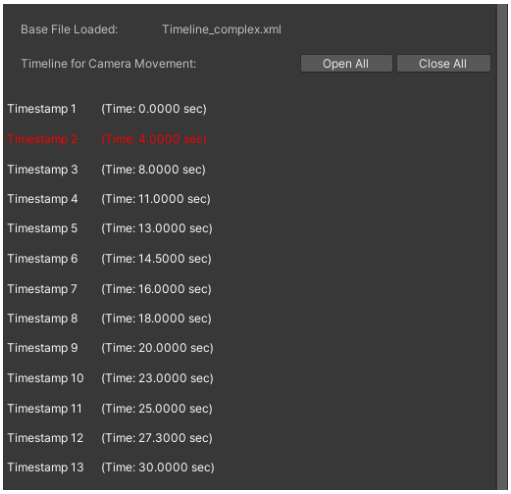


Figure 45: Collapsed Timestamps

The last button that has not been discussed is the “Start Simulation” button. This button is not visible during the editing mode as the simulation will not be running in the edit mode. Although all the functionality explained thus far can be used during the edit mode and play

mode, the “Start Simulation” button will only appear once in play mode. This button can be seen in **Figure 46**. For this portion of the simulation, this button will use real time to increment the “Current Time in Simulation” through the time range of timestamps.

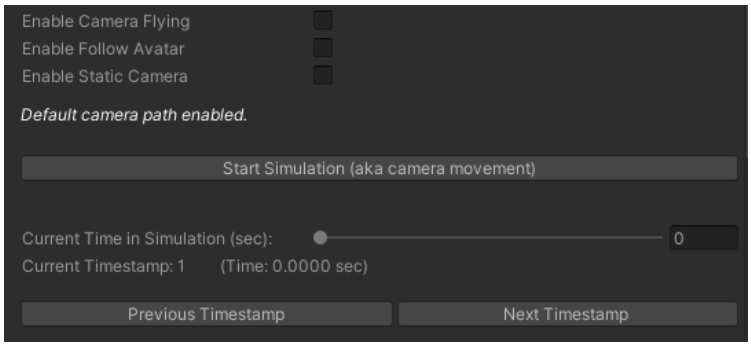


Figure 46: “Start Simulation” Button when in Play Mode

After starting the simulation, two options will appear in place of the “Start Simulation” button: “Pause Simulation” and “Stop Simulation.” The “Stop Simulation” button stops the simulation at the current time, which results in the camera staying in the current location. “Pause Simulation” button will stop incrementing the current time until either the user selects the “Continue Simulation” button or the “Stop Simulation” button. These buttons can be seen in **Figure 47** and **Figure 48**.

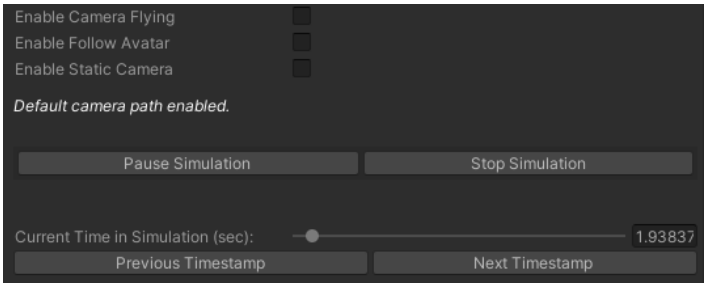


Figure 47: Simulation Currently Running

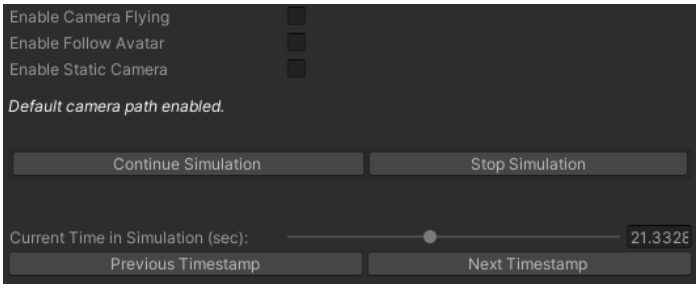


Figure 48: Simulation Currently Paused

4.2 Optimized Camera Path

The C# scripting language has a class known as “AnimationCurve.” This class stores a collection of “Keyframes” that can be evaluated over time. Keyframes are similar to what is defined in this paper as the Timestamp but keyframes contain only a single time and a single value (e.g. only the x value of the location), while the timestamps contain more information such as the full location, the full orientation, and the follow location/orientation of the avatar. These keyframes are able to be used in an animation curve but for this simulation, there will need to be six different lists of keyframes: x location, y location, z location, x rotation, y rotation, and z rotation.

From here, the animation curve class has a method called “SmoothTangents” which will use the defined keyframes to create a smooth curve by smoothing the in and out tangents for each keyframe. This method defines a mathematical function which can be evaluated at any time value. To create a visual curve for the user, the segments between the keyframes can be divided into smaller increments and evaluated in the animation curve. Since this is done for each list of keyframes, the evaluations results can then be combined and saved into arrays of values: one for the location (x, y, and z) and one for the rotation (x, y, and z). The array of the locations for the full curve can be visualized in the scene view. (Haagndaaz, 2016)

The process described works without any further corrections for the location, but for the rotation, a correction is required prior to the smoothing of the tangents. Because of how the angles of the rotations are defined, if the rotation on an axis was at 350 degrees for example and needed to go to 10 degrees, the logical way to do this would be to go to 360 degrees which equals 0 degrees then to 10 degrees, but due to the smoothing of the tangents, it tries to rotate all the way back around through 180 degrees and then to 10 degrees. To avoid this, an algorithm was created to avoid a difference of 180 degrees between two keyframes. For the example previously given, a rotation from 350 degrees to 10 degrees is actually only a 20 degree difference but when the math is done of 350 – 10 = 340, this would indicate that a change to the 10 degree keyframe needs to be made as 340 is greater than 180. This is simply done by adding 360 degrees to all future values (which only changes the number of the angle but not the actual orientation), resulting in the new values of 350 and 370 degrees. The same method is implemented if going from 10 degrees to 350 degrees: 10 – 350 = -340.

In this case, it is less than -180 and so 360 is subtracted from all future values including 350, resulting in -10, so the new points are 10 and -10 degrees. By checking this for every value, corrections can be made to the rotations to ensure the quickest path is taken to the target orientation. This correction must be done for each rotational axis (x, y, and z). This procedure can be visualized using **Figure 49**.

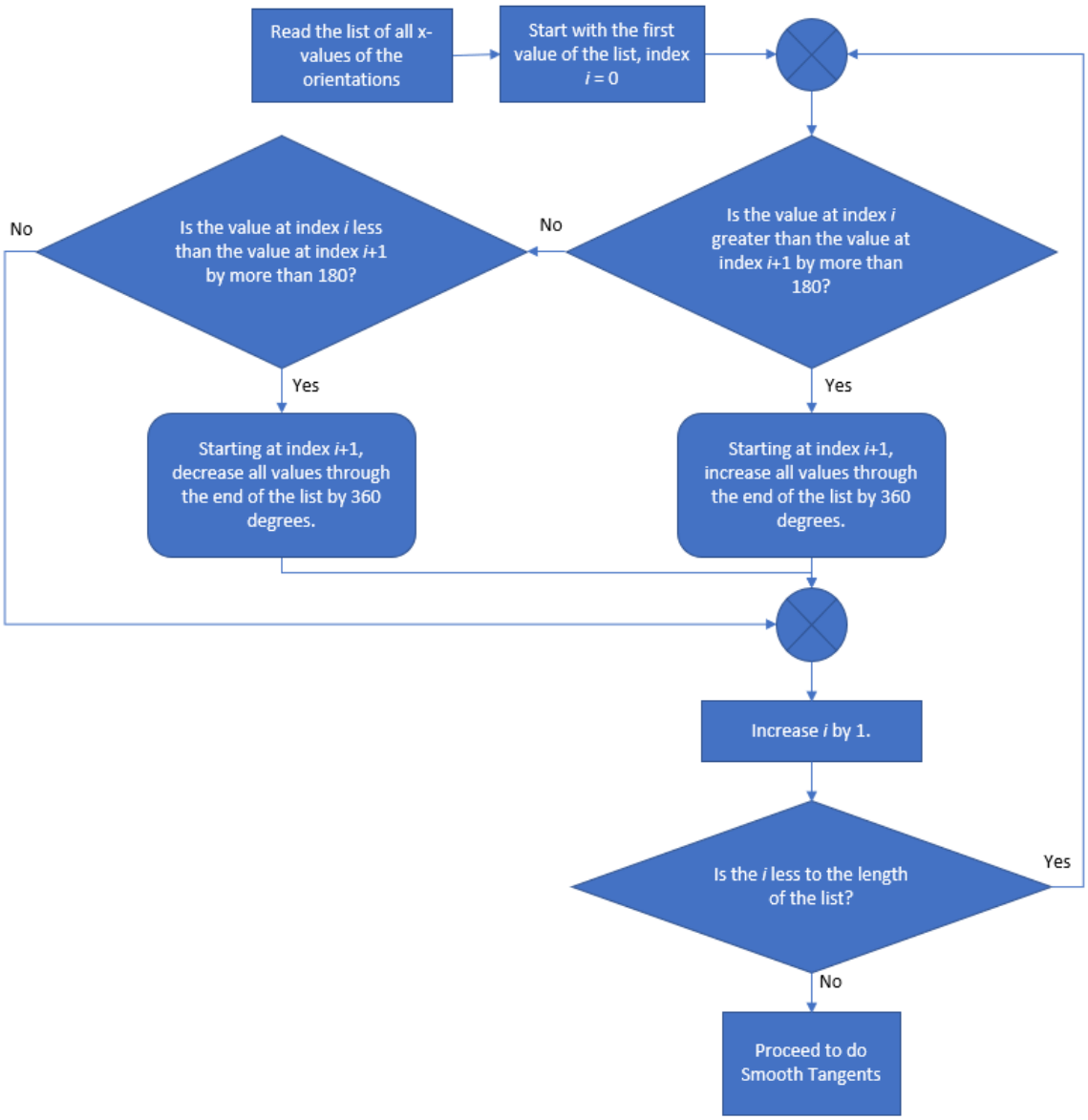


Figure 49: Logic Flowchart of Rotational Correction of Keyframes

5 RESULTS

When trying to display the results of the view orientation optimization procedure, it is not feasible to display all the possible images produced from each view and the scores associated with those images. Instead, there were a few parts selected and a few of the best images for those parts will be presented for both methods: weighted scores and *k*-NN. Although these procedures work for any object, the procedures will only be used in the MOSIM project to analyze the objects defined as "Parts" in the scene. Example parts that were analyzed include the Steering Wheel, Pedal, Chair, and Mainframe, but the Fixture was also analyzed, even though it is not a part. These objects have varying complexity to show that these procedures work with any object of any shape or size with any feature. Other simple geometrical objects can be analyzed, too, such as a cube, torus, or sphere, which is unique as all views are the same so there is no best view.

To determine the best views for these parts for weighted scores, as previously mentioned, the plan was to use Unity Simulation (or local device) to cycle through varying weight combinations in order to determine the optimal weight combination for the weighted scores. Although this was the original plan, it seemed that by cycling through all the weights, the optimal weight combination typically consisted of one metric being 100% while all the rest were 0%. The method was adjusted multiple times with different maximum percentages allowed per metric (e.g. 75%, 50%, 35%) and different increments for the weights (e.g. 0, 5, 10…; 0, 10, 20…; etc.) but the weight combinations still consisted of certain weights being maxed out with others being 0% or 5% / 10%, depending on the weight increments. Because of this, a weight combination was manually chosen by visually inspecting different weight combinations to see which produced reasonable results. The chosen weight combination that was used in this section and the discussion section is shown in Table 4, along with the parameters for the options used during the testing. The number of best view pictures is listed as *Z* because in this testing, all the views had pictures taken to obtain specific views.

Table 4: Parameters and Weights used for Weighted Scored and k-NN

Options	**Value**
Number of Pictures	12 (yaw), 5 (pitch), 5 (roll)
Acceptable Gray Range	0.05
Acceptable Pixel Match Ratio	0.75
Exception Pixel Match Ratio	0.9
Pixel Skip Size	2
Screenshot Scale	0.5
Number of Best View Pictures	*Z*
Parameters	**Chosen Weights**
Projected Area	0.10
Visible Surface Area	0.15
Center of Mass	0.10
Symmetry	0.30
Visible Edges	0.25
Number of Mesh Triangles	0.10

5.1 Metrics Comparison

Figure 50 through **Figure 70** display the best, worst, and intermediate view for each metric for the "Mainframe" along with the raw (not normalized) value of the respective metric. Only one part was analyzed in this section of the results as to reduce the number of images presented. This is completed by using the options in *Table 4* and by setting the metric's weight to one while setting the other metrics to zero, thereby only analyzing the part with that specific metric. The results may vary slightly from user to user, though, as the analysis of the parts for some metrics rely on the resolution of the game view, and so, some views may be scored higher or lower.

Figure 50: Projected Area – Worst Raw Value (Value: 0.01725425)

Figure 51: Projected Area – Intermediate Raw Value (Value: 0.04296296)

Figure 52: Projected Area – Best Raw Value (Value: 0.0545809)

Figure 53: Visible Surface Area – Worst Raw Value (Value: 0.1495513)

Figure 54: Visible Surface Area – Intermediate Raw Value (Value: 0.3858489)

Figure 55: Visible Surface Area – Best Raw Value (Value: 0.4807692)

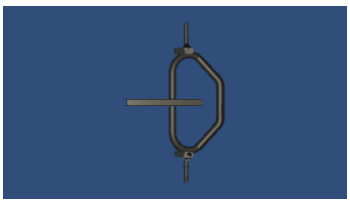


Figure 56: Center of Mass X – Worst Raw Value (Value: 0.1157713)

Figure 57: Center of Mass X – Intermediate Raw Value (Value: 0.04356625)

Figure 58: Center of Mass X – Best Raw Value (Value: 7.258441E-05)

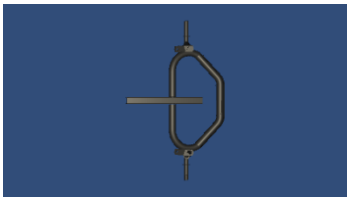


Figure 59: Center of Mass Y – Worst Raw Value (Value: 0.4127108)

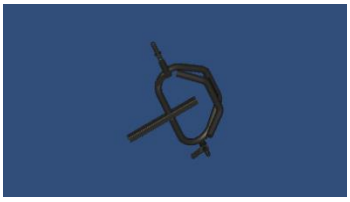


Figure 60: Center of Mass Y – Intermediate Raw Value (Value: 0.2905233)

Figure 61: Center of Mass Y – Best Raw Value (Value: 0.06499785)

Figure 62: Symmetry – Worst Raw Value (Value: 0.6380546)

Figure 63: Symmetry – Intermediate Raw Value (Value: 0.215781)

Figure 64: Symmetry – Best Raw Value (Value: 0.1209455)

Figure 65: Visible Edges – Worst Raw Value (Value: 0.006326928)

Figure 66: Visible Edges – Intermediate Raw Value (Value: 0.01358953)

Figure 67: Visible Edges – Best Raw Value (Value: 0.01742133)

Figure 68: Mesh Triangles – Worst Raw Value (Value: 0.1581126)

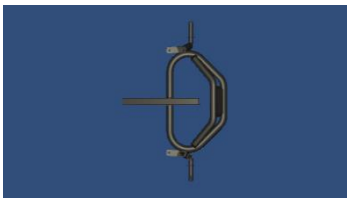


Figure 69: Mesh Triangles – Intermediate Raw Value (Value: 0.3836921)

Figure 70: Mesh Triangles – Best Raw Value (Value: 0.4958609)

In *Table 5*, the values for the views in **Figure 50** through **Figure 70** are presented under the Mainframe to demonstrate the range of the values for each metric. For the Pedal and Fixture, the values were also obtained in the same manner as the Mainframe to compare multiple object's range of values for each metric. A visualization of the differences of these ranges can be seen in the plot in **Figure 71**.

Table 5: Example Value Ranges for Metrics of Different Objects

	Mainframe	Pedal	Fixture
Projected Area	(0.01725 – 0.05458)	(0.04700 – 0.15610)	(0.03139 – 0.08164)
Visible Surface Area	(0.14955 – 0.48076)	(0.08876 – 0.36043)	(0.09284 – 0.34361)
Center of Mass X	(0.00007258 – 0.11577)	(0.0001337 – 0.1602694)	(0.00153 –0.12770)
Center of Mass Y	(0.06499 – 0.4127)	(0.09102 – 0.55716)	(0.18465 –0.45189)
Symmetry	(0.1209 – 0.6380)	(0.1008 – 0.9698)	(0.04114 –0.91599)
Visible Edges	(0.00633 – 0.01742)	(0.00355 – 0.01846)	(0.00829 – 0.02853)
Number of triangles	(0.15811 – 0.49586)	(0.07000 – 0.41333)	(0.14655 – 0.42959)

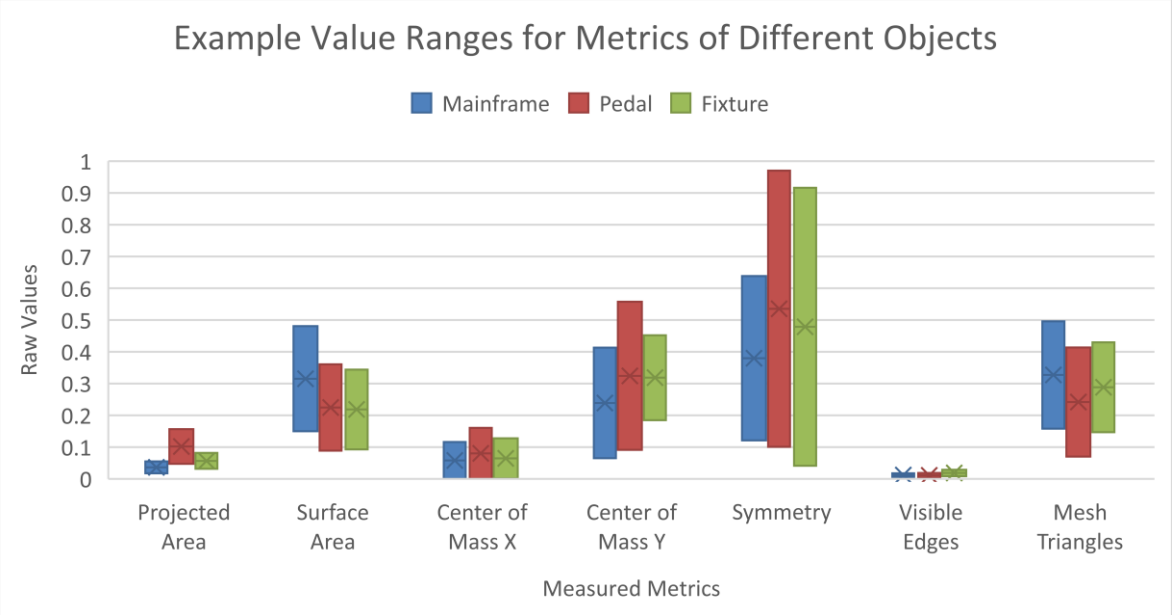


Figure 71: Example Value Ranges from *Table 5* Plotted

5.2 Parts Comparison

Using the options and parameters from *Table 4* in combination with the training data set for *k*-NN, each of the parts in the following sections were analyzed and for each part, certain views, along with the respective score, are presented. For each part, the goal was to present the best three and worst three screenshots for each method (weighted scores and *k*-NN), but if similar views from previously presented screenshots appear in the next screenshot as well, it will be skipped so a noticeably different screenshot can be presented and labeled with the respective view number.

5.2.1 Steering Wheel

Figure 72 and **Figure 73** show the best views and the worst views of the Steering Wheel part using the manually selected weights.

(a) 1st Best View

(Score: 0.8181105)

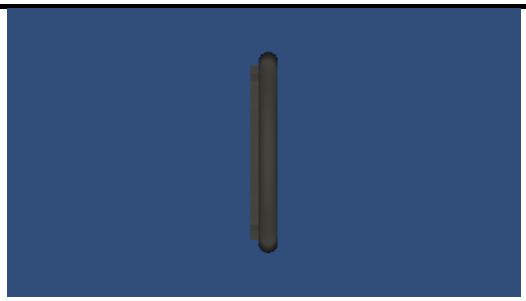
(b) 2nd Best View

(Score: 0. 8045228)

(c) 3rd Best View

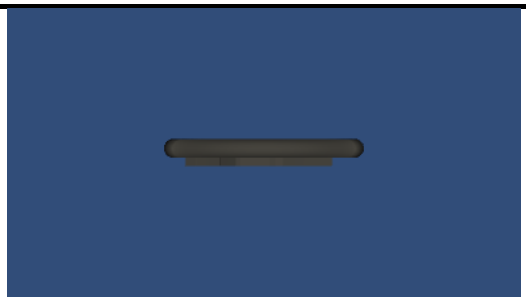
(Score: 0.79085)

Figure 72: Steering Wheel – Weights – Best Views



(a) 1st Worst View

(Score: 0.1682093)



(b) 9th Worst View

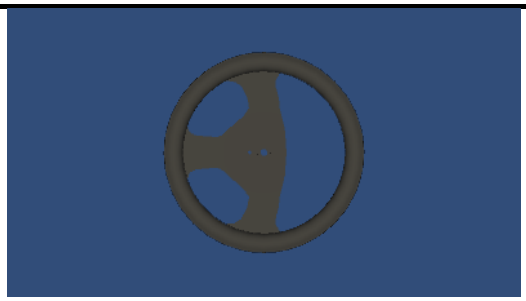
(Score: 0.1795437)

(c) 89th Worst View

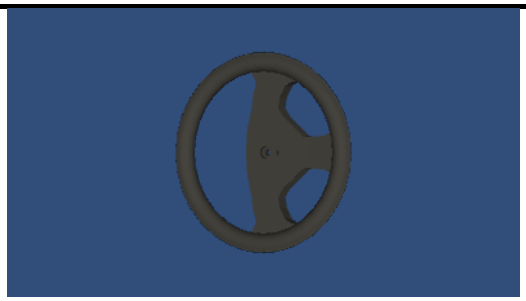
(Score: 0.5215901)

Figure 73: Steering Wheel – Weights – Worst Views

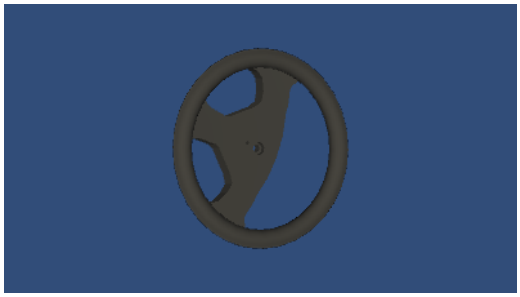
Figure 74 and **Figure 75** show the best views and worst views of the Steering Wheel using the *k*-NN algorithm.



(a) 1st Best View
(Score: 9.904285)

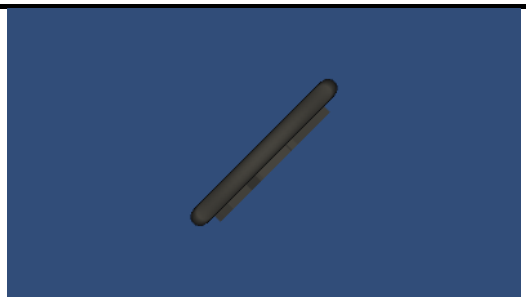


(b) 12th Best View
(Score: 10.24834)

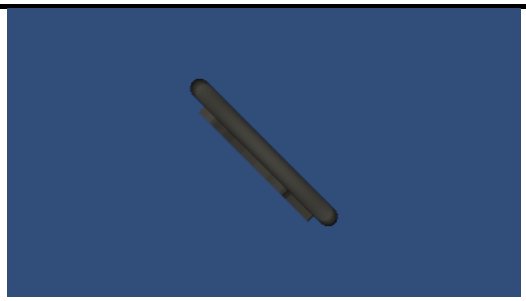


(c) 17th Best View
(Score: 10.34186)

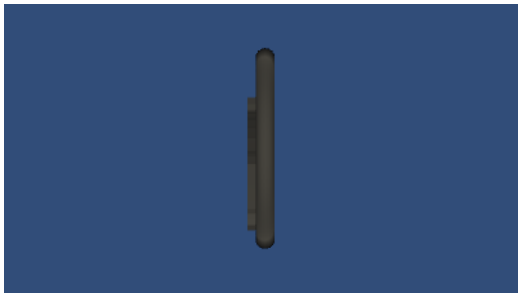
Figure 74: Steering Wheel – *k*-NN – Best Views



(a) 1st Worst View
(Score: 19.00189)



(b) 3rd Worst View
(Score: 18.6986)



(c) 9th Worst View

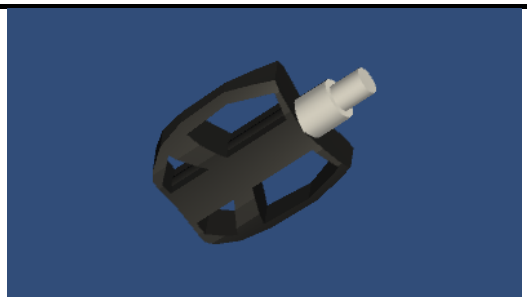
(Score: 18.04977)

Figure 75: Steering Wheel – *k*-NN – Worst Views

5.2.2 Pedal

Figure 76 and **Figure 77** show the best views and the worst views of the Pedal part using the manually selected weights.

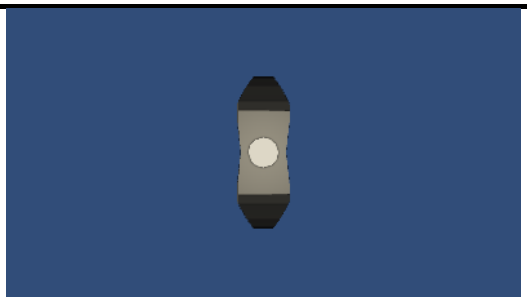
(a) 1st Best View
(Score: 0.87352)



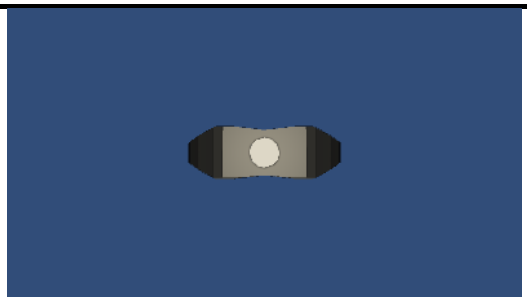
(b) 2nd Best View
(Score: 0.8732207)

(c) 3rd Best View
(Score: 0.8722191)

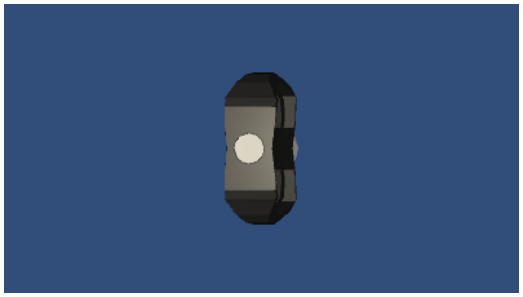
Figure 76: Pedal – Weights – Best Views



(a) 1st Worst View
(Score: 0.08299988)



(b) 2nd Worst View
(Score: 0.1012878)

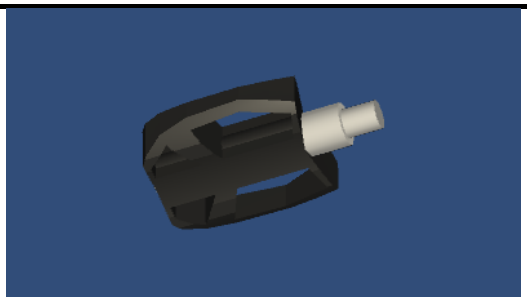


(c) 7th Worst View

(Score: 0.3549161)

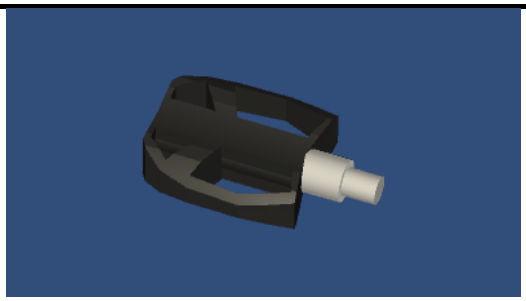
Figure 77: Pedal – Weights – Worst Views

Figure 78 and **Figure 79** show the best views and worst views of the Pedal using the *k*-NN algorithm.



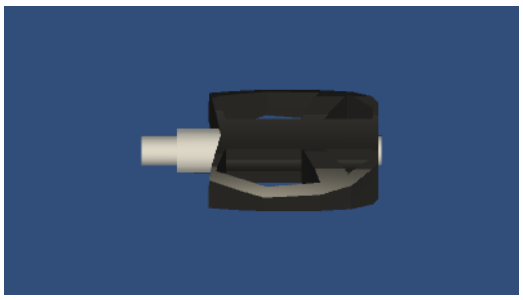
(a) 1st Best View

(Score: 10.42446)



(b) 2nd Best View

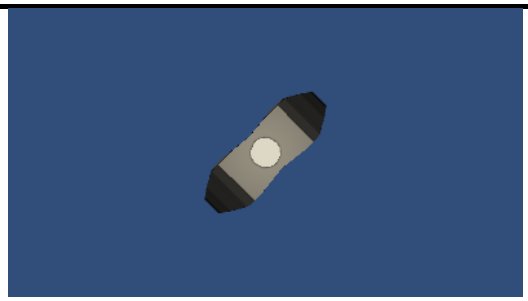
(Score: 10.52268)



(c) 3rd Best View

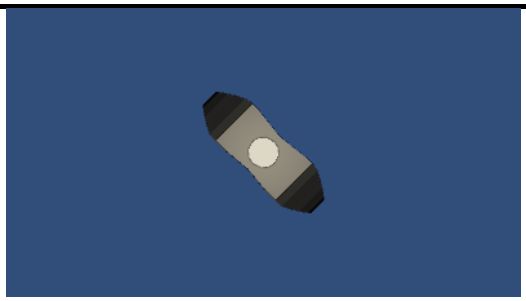
(Score: 10.71553)

Figure 78: Pedal – *k*-NN – Best Views



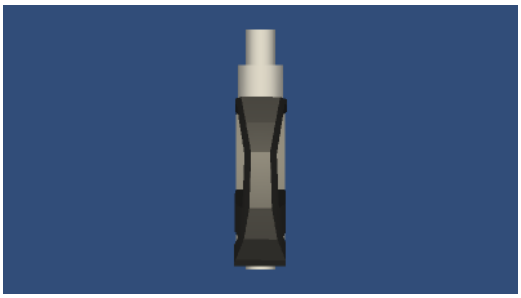
(a) 1st Worst View

(Score: 23.93457)



(b) 2nd Worst View

(Score: 23.84394)



(c) 3rd Worst View

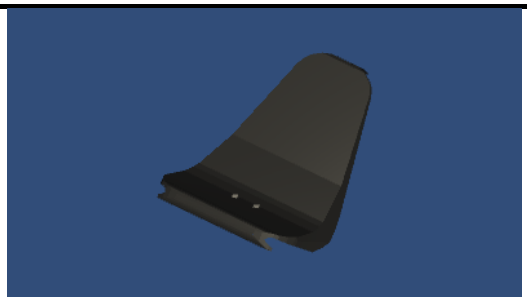
(Score: 23.36762)

Figure 79: Pedal – *k*-NN – Worst Views

5.2.3 Chair

Figure 80 and **Figure 81** show the best views and the worst views of the Chair part using the manually selected weights.

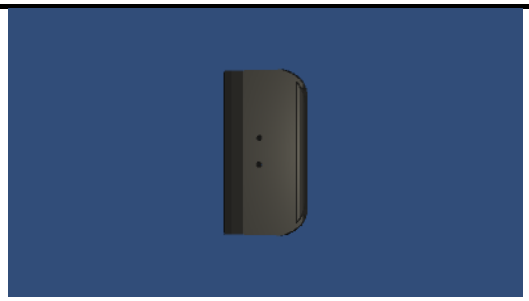
(a) 1st Best View
(Score: 0.8489632)



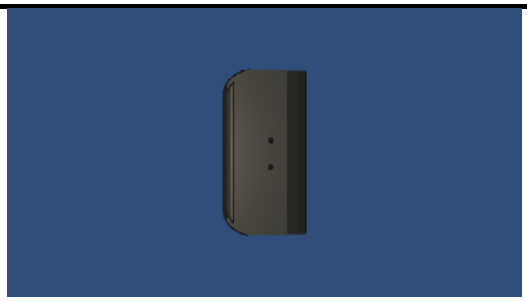
(b) 2nd Best View
(Score: 0.8067668)

(c) 3rd Best View
(Score: 0.7817746)

Figure 80: Chair – Weights – Best Views



(a) 1st Worst View
(Score: 0.2700411)



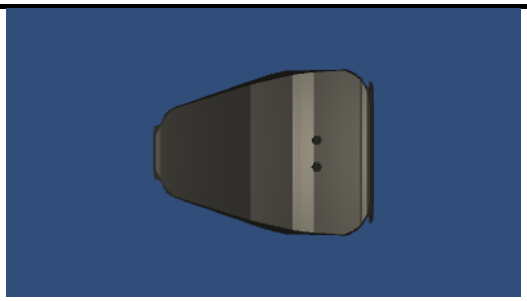
(b) 2nd Worst View
(Score: 0.3057504)

(c) 3rd Worst View

(Score: 0.31623)

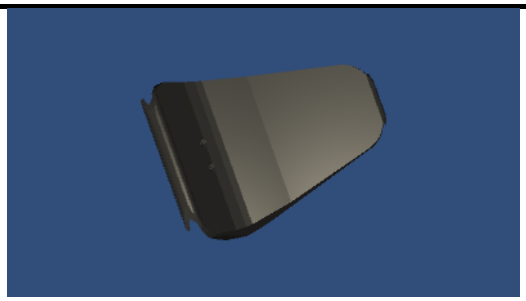
Figure 81: Chair – Weights – Worst Views

Figure 82 and **Figure 83** show the best views and worst views of the Chair using the *k*-NN algorithm.



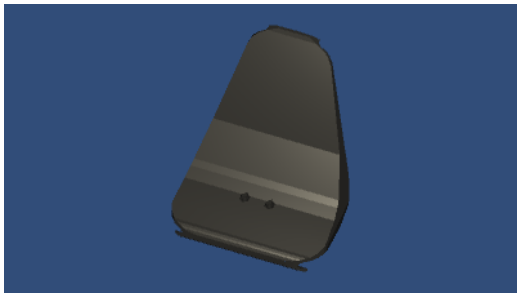
(a) 1st Best View

(Score: 10.40772)



(b) 6th Best View

(Score: 10.67428)



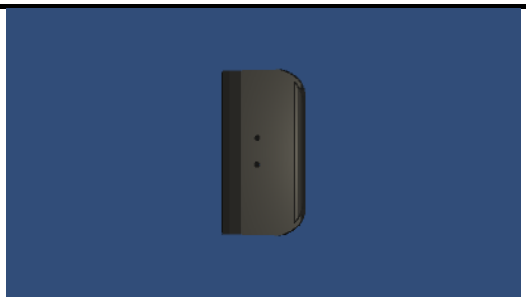
(c) 7th Best View

(Score: 11.20491)

Figure 82: Chair – *k*-NN – Best Views

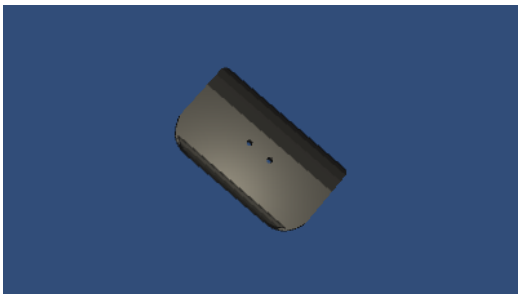
(a) 1st Worst View

(Score: 22.40473)



(b) 2nd Worst View

(Score: 21.27069)



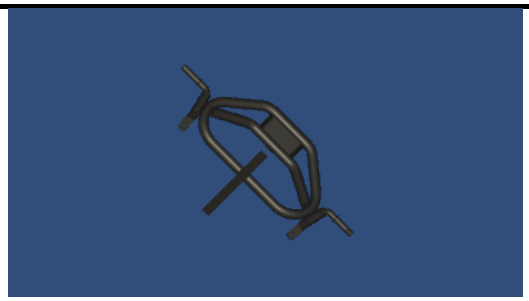
(c) 3rd Worst View

(Score: 21.26901)

Figure 83: Chair – *k*-NN – Worst Views

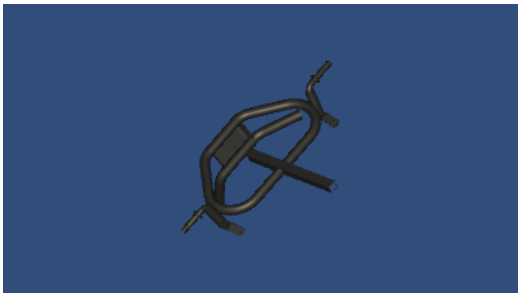
5.2.4 Mainframe

Figure 84 and **Figure 85** show the best views and the worst views of the Mainframe part using the manually selected weights.



(a) 1st Best View
(Score: 0.8455795)

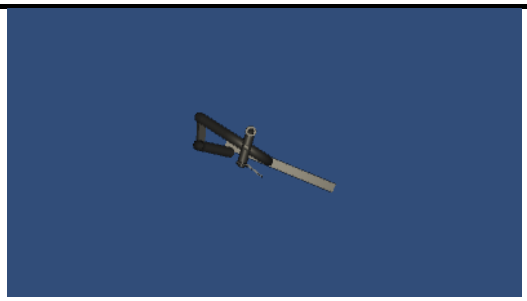
(b) 2nd Best View
(Score: 0.8371233)



(c) 3rd Best View
(Score: 0.8360713)

Figure 84: Mainframe – Weights – Best Views

(a) 1st Worst View
(Score: 0.33220392)



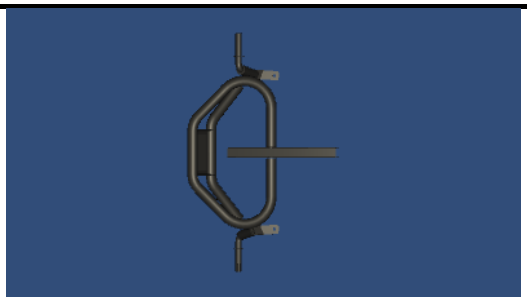
(b) 2nd Worst View
(Score: 0.3413007)

(c) 3rd Worst View

(Score: 0.3420706)

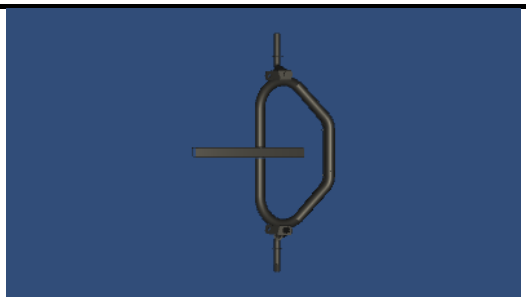
Figure 85: Mainframe – Weights – Worst Views

Figure 86 and **Figure 87** show the best views and worst views of the Mainframe using the *k*-NN algorithm.



(a) 1st Best View

(Score: 11.65027)



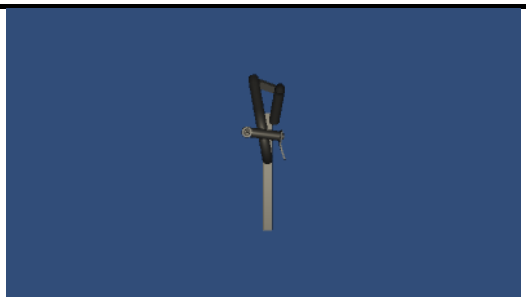
(b) 8th Best View

(Score: 12.13669)

(c) 12th Best View

(Score: 12.29754)

Figure 86: Mainframe – *k*-NN – Best Views



(a) 1st Worst View
(Score: 20.22642)

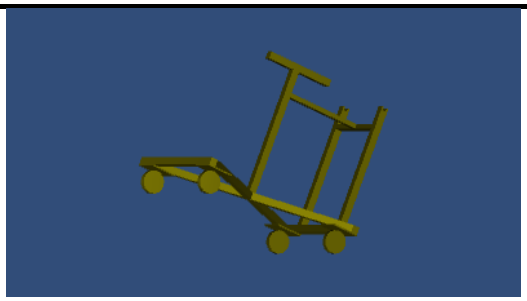
(b) 3rd Worst View
(Score: 19.95799)

(c) 5th Worst View
(Score: 19.51861)

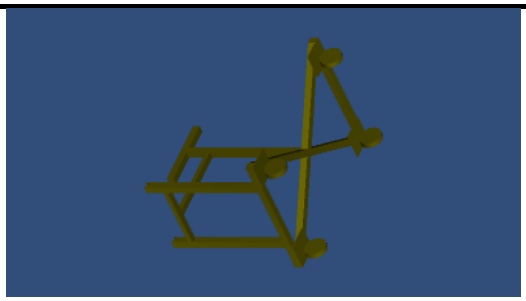
Figure 87: Mainframe – *k*-NN – Worst Views

5.2.5 Fixture

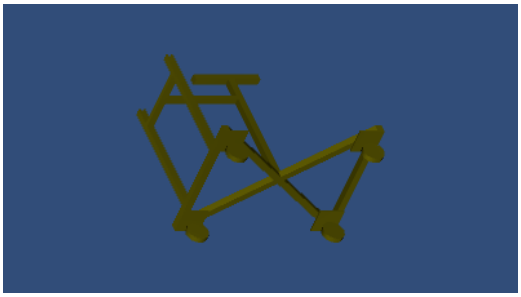
Figure 88 and **Figure 89** show the best views and the worst views of the Fixture part using the manually selected weights.



(a) 1st Best View
(Score: 0.786476)

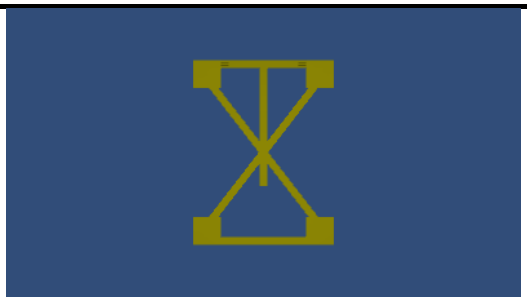


(b) 3rd Best View
(Score: 0.7480069)

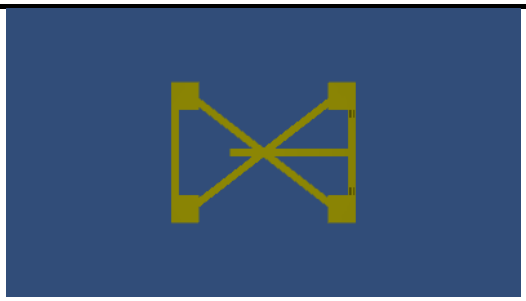


(c) 4th Best View
(Score: 0.746846)

Figure 88: Fixture – Weights – Best Views



(a) 1st Worst View
(Score: 0.1947843)



(b) 2nd Worst View
(Score: 0.199606)

(c) 4th Worst View

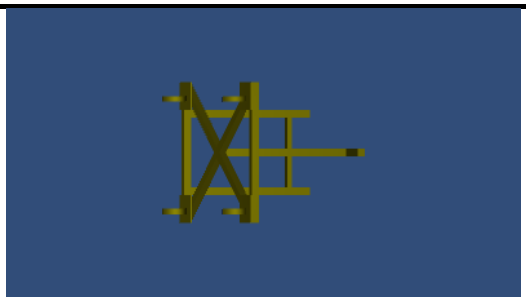
(Score: 0.2831084)

Figure 89: Fixture – Weights – Worst Views

Figure 90 and **Figure 91** show the best views and worst views of the Fixture using the *k*-NN algorithm.

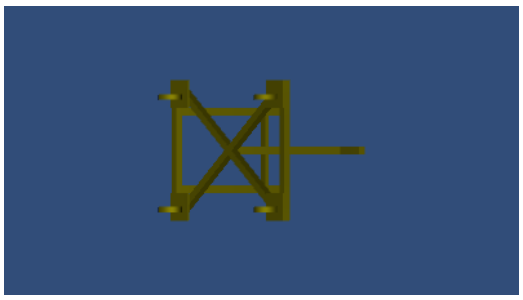
(a) 1st Best View

(Score: 12.19623)



(b) 3rd Best View

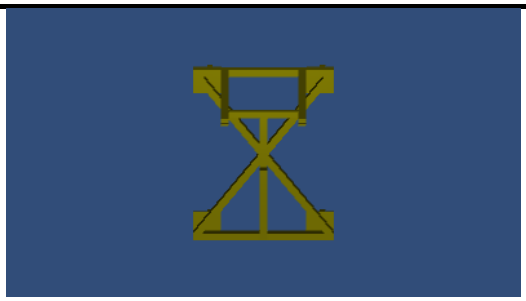
(Score: 12.32406)



(c) 4th Best View

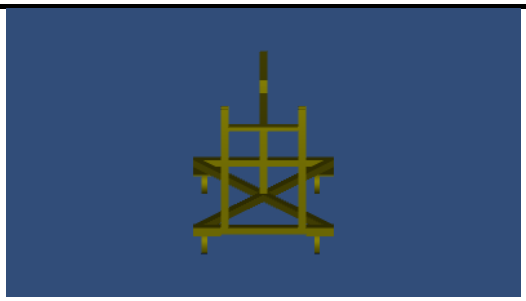
(Score: 12.3761)

Figure 90: Fixture – *k*-NN – Best Views



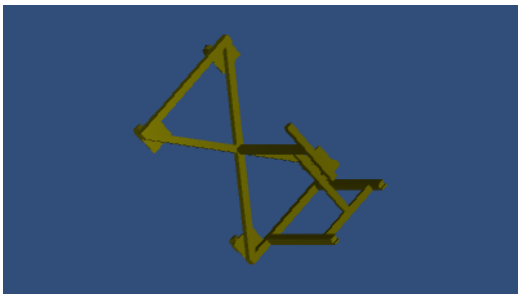
(a) 1st Worst View

(Score: 21.09947)



(b) 3rd Worst View

(Score: 20.46036)



(c) 5th Worst View

(Score: 20.09161)

Figure 91: Fixture – *k*-NN – Worst Views

5.3 Camera Path Results

The results for the camera path are best defined through the camera's view during the simulation. Although a video would be best to present this, an alternative method for presenting the results is used. **Figure 92** displays all the timestamps used in this example for the camera path. **Figure 93** displays how the camera's path is defined for a varying number of timestamps. In these figures, the path shown is only displaying the path for the XZ-plane but in the actual camera path, it is also done for the XY-plane. The red line represents the path if a direct line is drawn between the points, while the black line represents the smooth path of the camera that is created from the camera path optimization utility. In this example, points are added in the middle of the list to create a path around the avatar and fixture.

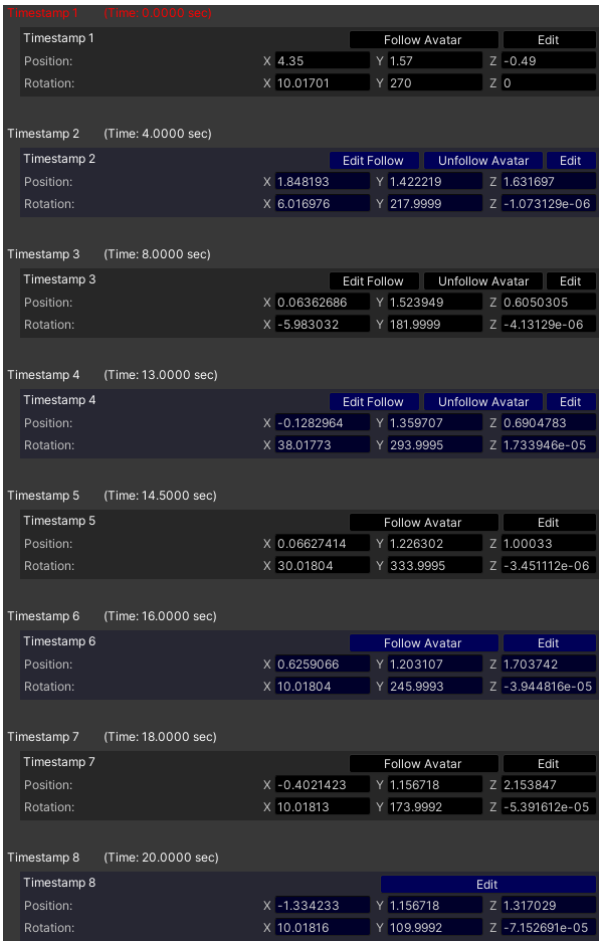
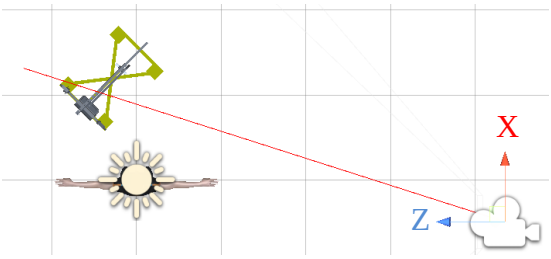
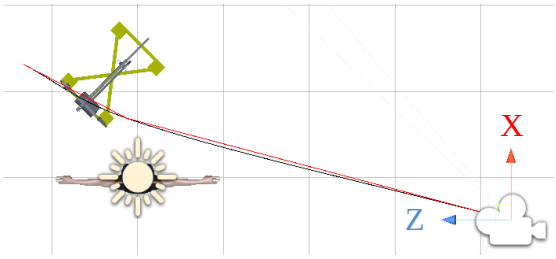


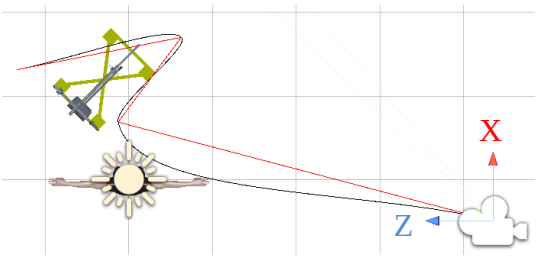
Figure 92: Full List of Timestamps



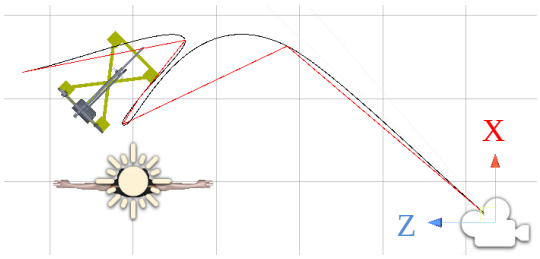
(a) 2 Timestamps: 1 and 8



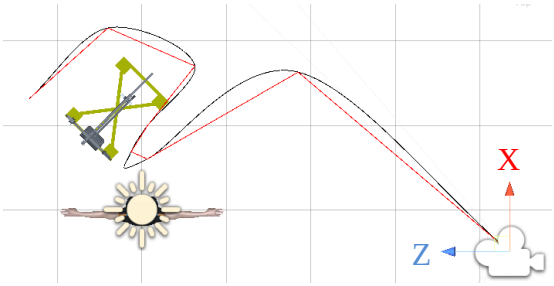
(b) 3 Timestamps: 1, 4, and 8



(c) 4 Timestamps: 1, 4, 6, 8



(d) 6 Timestamps: 1, 2, 4, 5, 6, 8



(e) All 8 Timestamps

Figure 93: Camera Paths with Different Number of Timestamps

In addition, the rotation of the camera is not easily visualized except through still images during the simulation. Because of this, **Figure 94** is used to demonstrate the motion of the camera (position and rotation) through a timeline of images.

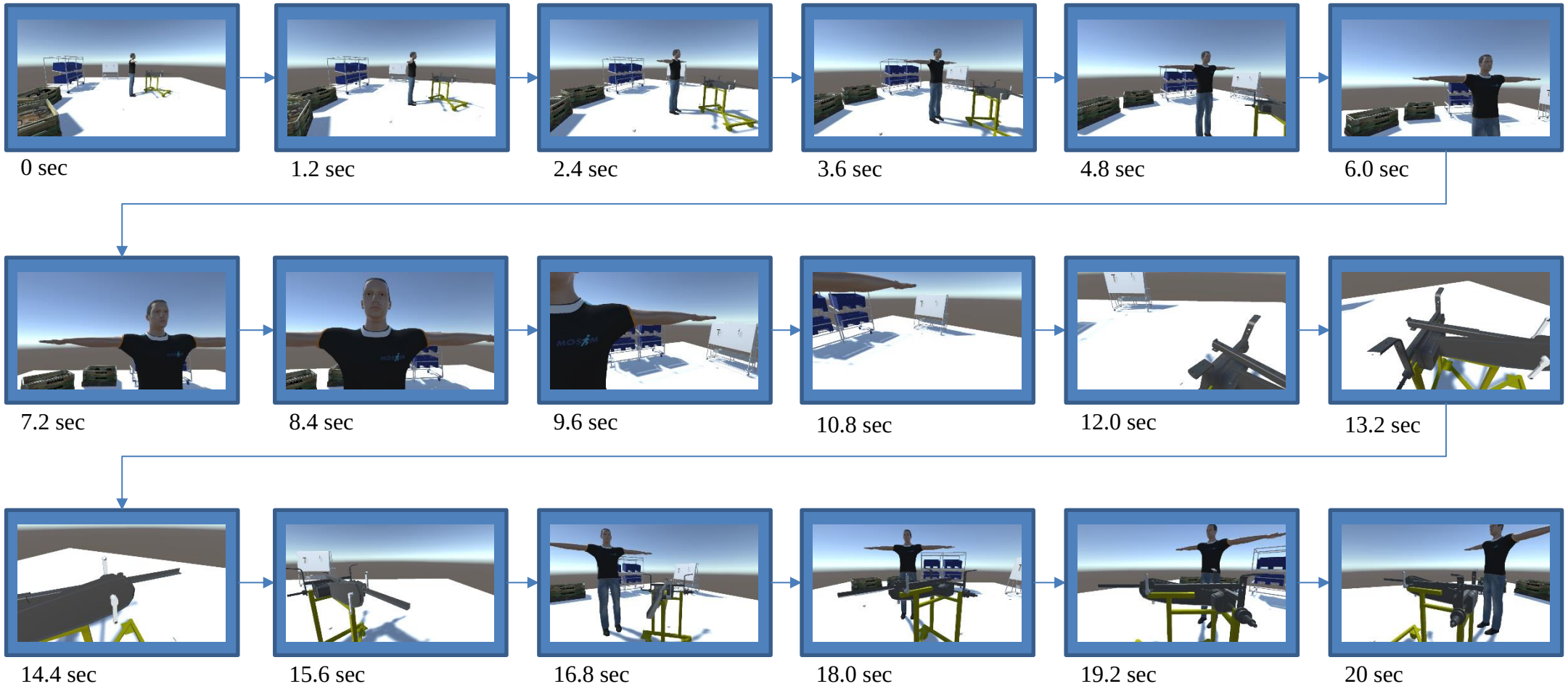


Figure 94: Camera Path Demonstration Timeline

6 DISCUSSIONS

Prior to reviewing the results presented in Section 5, it is necessary to talk about the implementation of the research in Unity. As mentioned, Unity is the target engine of choice in this work as Unity is the target engine that was already being used for this use case for MOSIM. Successful implementation of the analysis of the parts in Unity required that specific properties of the parts be analyzed and verified to ensure that the calculated values be logical for the given part.

For all the metrics, the values at varying views were compared to what would logically be assumed that the value should be and compared to other objects by comparing the magnitude of the measured values. For example, for the percentage of visible surface area, unless the part has many recessed areas or hidden features, the percentage should rarely be over 50% as the user would only ever see less than half or about half of the object's surface at any given time. Verification of the values were completed, and this helped ensure that the implementation of the methods was done correctly and successfully. This was important to note as even before this verification, the values seemed accurate but when looking further at the exact values for each metric, there were discrepancies that needed to be resolved. For example, with the center of mass in the y-direction, theoretically, there should be no negative values as the value is the distance to the center of mass from the bottom of the bounding box and the center of mass can never be outside the bounds of the object, but somehow there was one value for a particular object that was negative. By looking into the timing of bounding box calculations, an error was noticed where the item was adjusted to the center of the screen but the location of the bounding box was not readjusted, which adversely affected the center of mass distance in the y-direction.

In general, by logically analyzing the values for each metric and the produced image, verification for the calculations of each metric was successfully completed and allowed for a display of the best, worst, and intermediate views as seen in **Figure 50** through **Figure 70**, which were previously presented. For each of the views presented in **Figure 50** through **Figure 70**, the views seem to accurately represent the corresponding metric. Some may seem

debatable, though, like in **Figure 55**, where the best view for the surface area seems like it does not have a lot of surface showing, but because of the specific features on the mainframe (i.e. the flat plate on the front), there is more surface in that view than any other view.

Another important detail to note is the range of the values as presented in *Table 5* and **Figure 71**. These display the importance of the normalization of the metrics. By setting each metric's minimum and maximum to zero and one respectively, the metrics can be fairly compared. Otherwise, for example, the mainframe's value for the visible surface area and the center of mass in the x-direction would result in completely different contributions to the final score, even when they have the same weight. *Table 5* and **Figure 71** effectively prove the need for normalization as without it, the metrics would not be weighed successfully.

6.1 Weighted Metrics / *k*-NN Algorithm

While implementing the methods in Unity, most of the testing was completed on a set of parts in the scene. In the results, a different set of items was chosen to prove that the methods can be used with a variety of parts. As mentioned previously, while presenting the results, the goal was to present the best three views and the worst three views, but since some of the views were too similar, they were skipped and the next view was checked to see if it is different enough. This was done in hopes that it would provide a more accurate image of how the best and worst views varied.

When looking at the best views for the weighted metrics, every part seems to agree that the best views tend to be at varying isometric views, which is when a 3D object is displayed in two dimensions. The real discrepancy lies in the fact that some may be over-rotated or that the isometric view is at an angle at which the part is not typically viewed. The Steering Wheel (**Figure 72**), Pedal (**Figure 76**), Chair (**Figure 80**), and Mainframe (**Figure 84**) all have their top views as views that could be considered preferred, but the Fixture (**Figure 88**) only has its 1st best view as a logical orientation while the other two presented are less optimal. On the other hand, when reviewing the worst views for the same objects, Steering Wheel (**Figure 73**), Pedal (**Figure 77**), Chair (**Figure 81**), Mainframe (**Figure 85**), and Fixture (**Figure 89**), they all tend to be some orthographic view (e.g. top, side, front, etc.) of the object. This proves the idea that the isometric views really do show more detail of the

part than orthographic views, but the comparisons between the isometric views need to be considered. Certain isometric views are very similar but rotated in such a way that the part is no longer in an orientation is that considered “normal” by the user.

Moving to the *k*-NN algorithm results, these results are different from the weights in many ways. Firstly, the scores for these images may seem different from the logic in Section 3.4 as the lowest scores were chosen for the best views instead of the highest scores. The implementation in Unity followed the logic from Section 3.4 but the results seemed to be better when searching for the lowest score versus the highest score. This deviation from Section 3.4 could be due to the heavy reliance on the training data selected and the fact that for these results only 18 data points were manually selected (3 views for each of 6 parts). Further investigation into this could be done in the future when more data points are added. In addition, the results for *k*-NN are not as uniform as the weighted scores. For example, for the Steering Wheel in **Figure 74** and the Chair in **Figure 82**, view (a) for both is an orthographic view, even though they are considered the 1st best views by the algorithm, while (b) and (c) are more isometric views. This result could be further improved by selecting more training data points for a wide variety of parts and objects, which would create an even broader range of values from which to calculate the score of the view. The Mainframe in **Figure 86** is a similar case where (a) and (c) are both isometric and (b) is orthographic. Although slightly different from the Steering Wheel and Chair, the Mainframe suffers from the same case where an orthographic view is considered one of the best views, even if it is generally not preferred. The Pedal in **Figure 78** seems to be the only case from these test objects that have all three of its best views as isometric views and that those isometric views are presented in an acceptable orientation. This could be due to its more complex geometry in the form of holes and circular features. Meanwhile, the Fixture in **Figure 90** is a very unique case as the algorithm did not choose simply orthographic views or isometric views, but a combination between the two, as all three best views seem to be in between two orthographic views, but does not quite incorporate the third rotation to make it fully isometric.

In contrast, the worst views are worse than the best view results as they should be but not by much. For the Steering Wheel in **Figure 75** and Pedal in **Figure 79**, all of the views are

orthographic views which essentially present the Steering Wheel and Pedal with no indications of the detail of the object. For these parts, the objects are not easily identifiable from the views, which makes sense as these are supposedly the worst views. The Chair in **Figure 83** presents an isometric view for (a), but orthographic for (b) and (c). In (a), since it more of the bottom side of the Chair, it is also not distinguishable as to which part this might be. The Mainframe in **Figure 87**, similar to others, have two orthographic views for (a) and (b) but is between two orthographic views for (c), which provides some more detail but not much, which is as expected. Finally, the Fixture in **Figure 91** is probably the most interesting of the worst views from all the parts as (a) and (b) are between two orthographic views again, but (c) is an isometric view which actually provides a large amount of detail of the part. The only problem with (c) is that the orientation is not a logical orientation for the Fixture, which is expected since it is one of the worst views. In general, the difference between the best and the worst views for the *k*-NN algorithm is not large and the results overlap for some of them. This is most likely due to the fact that the *k*-NN algorithm is a form of ML and better results can be expected from an increase in the training data size.

When comparing the two methods of the best view orientation, in the current state, it appears that the weighted metrics come out ahead because of the consistency of the results. When comparing the best views of any object, the weighted metrics tend to have the orientation that is most preferred by users, except in the case of the Pedal where the results for the *k*-NN algorithm seemed decent and could be used as the screenshot for the Pedal. This is backed by a survey on Google Forms that was conducted on October 25-26, 2020 to see which screenshots were generally preferred. This survey was conducted on a sample size of 97 participants which were gathered using primarily social media (e.g. Facebook, Instagram, and direct messaging), which means that the participants come from a variety of educational backgrounds but are mostly consisting of friends and family. The results of this survey are shown in **Figure 95** through **Figure 100**, with the full survey (including the images) shown in Appendix II. For the survey, although the participants did not know which order the pictures were in and did not know what the pictures were for, they knew that they simply had to pick their preferred orientation of the part. The options in the survey consisted of "Orientation 1" and "Orientation 2" being the *k*-NN algorithm for every question and "Orientation 3" and "Orientation 4" being the Weight Metrics for every question. These

orientations came from the first two best images of the method shown for each object in the results in Section 5.2.

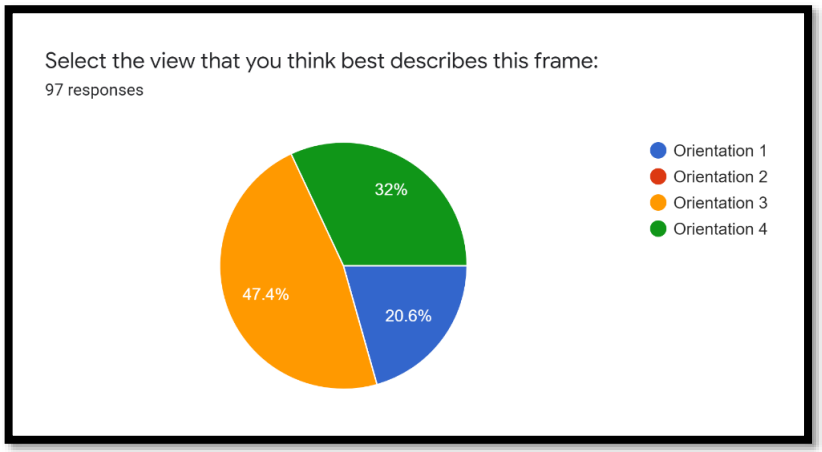


Figure 95: Survey Question Results – Mainframe

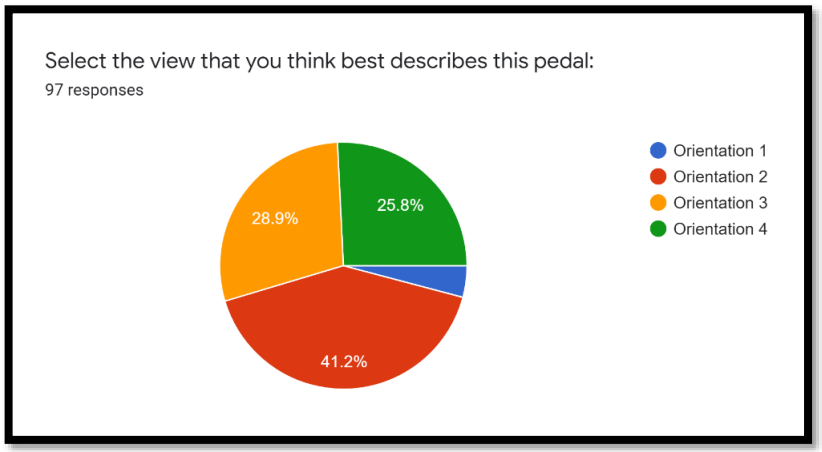


Figure 96: Survey Question Results – Pedal

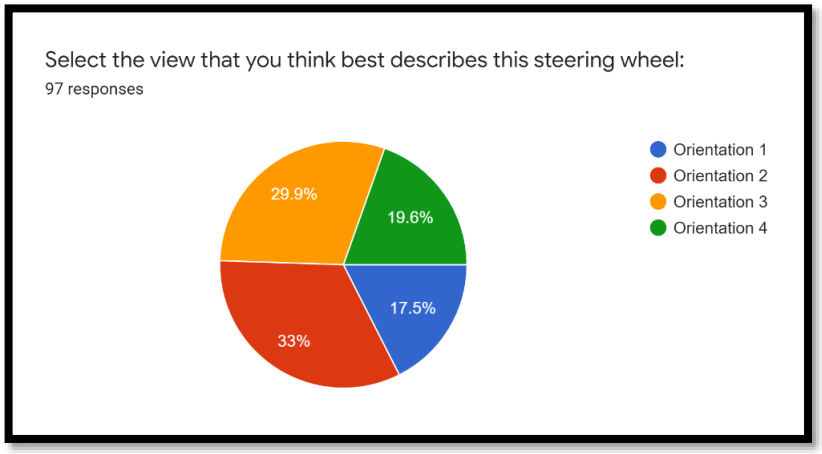


Figure 97: Survey Question Results – Steering Wheel

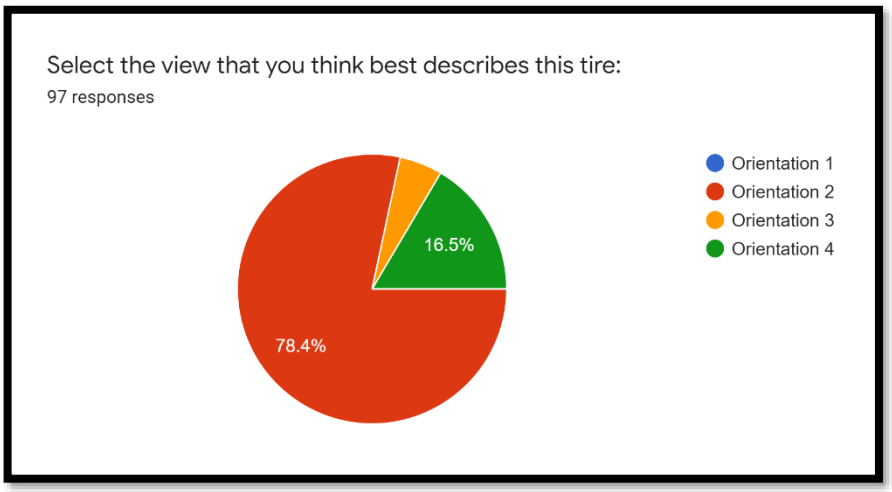


Figure 98: Survey Question Results – Tire

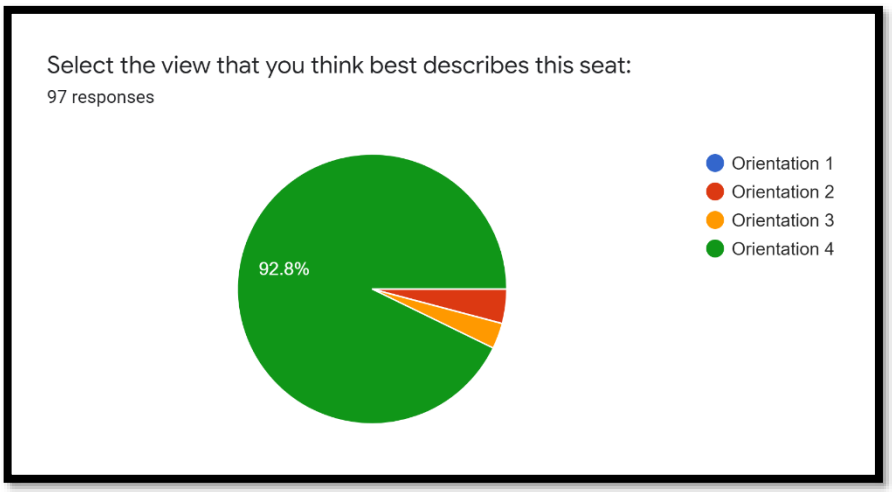


Figure 99: Survey Question Results – Chair

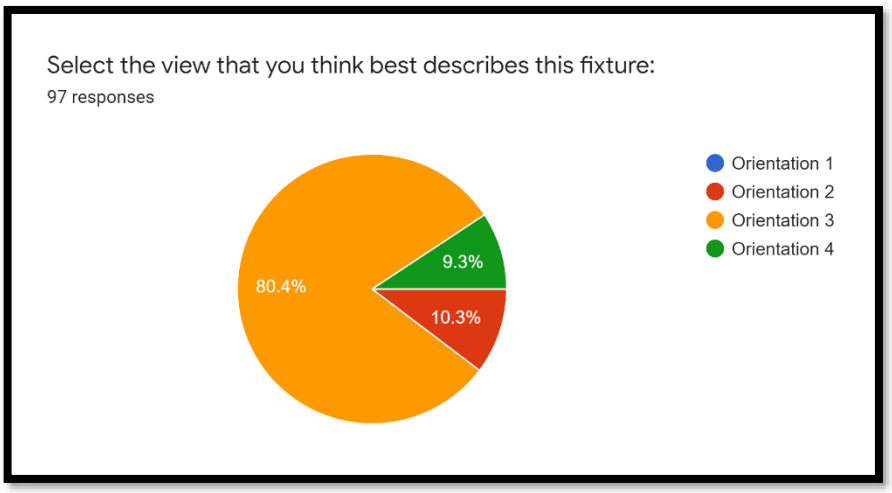


Figure 100: Survey Question Results – Fixture

When looking at the results of the survey, the first things that were noticed were the responses in the last question "(Optional) What was the main factor for your choices above?" which mostly concluded that the participants preferred to see parts from an isometric view. They preferred "an angle that represents more than one plane" and an angle that was "closest to the 'correct' orientation while the object [is] in use." Another point that participants noted was that there was some ambiguity about what some objects were. This is important to note because if the participant is not able to tell what the object is by a simple name and an image, the user in Unity will not be able to tell what the part is by a file name and thumbnail of the part. This means that the image is not sufficient for its purpose. Another comment was that the lighting on some of the objects made the image difficult to see which could cause another point of contention when deciding which view that they preferred.

In terms of which images were more preferred, when simply going by which image had the highest percentage of the vote, only two of the objects had an image for the k-NN algorithm that possessed the largest percentage of votes, while three of the objects had an image for the weighted metrics with the largest percentage. The Steering Wheel was a special exception because the views for the weighted metrics added up to about 50% and the views for the *k*-NN algorithm added up to about 50% as well, so the Steering Wheel was considered a tie for the participants. In general, though, the only object where the *k*-NN algorithm truly outperformed the weight metrics in the survey was for the Tire, which is not presented in the results section, but was included in this survey as an additional comparison because of how symmetric it is. Therefore, it could be concluded that in the current state of the project, the weight metrics generally outperform the *k*-NN algorithm.

6.2 Camera Path Utility

The camera path objective for this research was quite different as it relied more on the implementation than the research-side of it. From **Figure 92**, it is possible to see that the goal of being able to define custom timestamps was met as the user is able to create, delete, edit, save, load and clear the timestamps. In addition, it is possible to share the saved timestamps, which was tested by sending this code, along with the XML file to another user to preview the work. From **Figure 93** and **Figure 94**, the saved points were successfully used to create the path on which the camera would follow. Many options were incorporated

throughout the creation of this utility as new ideas would arise, and those ideas were considered particularly useful for the user. In general, this utility was created while keeping in mind that the user may not want both the photobooth utility and the camera path utility, and therefore, the system had to be coded in such a way that there would be no errors if the user only has one of them in their project. Particularly, the camera movement was the biggest concern as the camera utility wants to move the camera while in play mode and the photobooth will also try to move around the camera as well. A work-around had to be added to check which utilities are in the project to ensure that the proper utility had control of the camera and that the code would not try to access variables that do not exist. In the end, all the functions of this custom inspector and utility have been tested multiple times—throughout the creation of the utility and altogether once complete.

7 CONCLUSION

Throughout this work, the goal of the research was kept in mind to ensure that the work produced for MOSIM was complete. At the same time, the research necessary for the completion of the work involved both research surrounding the topic at hand and the software used throughout the project, specifically Unity. An important part of the research process is that it is with the intention of learning something new because if it is already known, there is no reason to complete the same work again. In the case of this work, not only was the topic of best view selection necessary but also learning how to use Unity.

The goals of creating an algorithm for selecting the best view and creating a utility for the camera path were both critical to the success of this research. Throughout the research for this project, the work evolved from only containing one method of calculating the best view, "Weighted Scores," to having a second possible method "*k*-NN Algorithm." These methods were created in the hopes of choosing one of the methods that can best select the best view for a wide variety of objects. In the current state of the project, after analyzing the results of both of the algorithms, the weighted scores seem to be more accurate and visually appealing to users but the hope is that over time, the *k*-NN algorithm will improve as more training data points are added to the list of data points and therefore, more objects will be able to be accurately analyzed. At the moment, from the survey and from the six objects analyzed during testing for the results, the Tire seems to be the only object for which the *k*-NN algorithm chose a better view. Because a variety of objects were chosen during implementation and a new set of objects were chosen during testing for the results, it allowed for a more accurate sense of which method was working better. In general, the *k*-NN algorithm is the method of choice in regards to the expected improvements that could be made over time, but the weighted scores is the method of choice in regards to the quality of the view chosen in the current state of the methods and data.

For the camera path utility, the goal was to make a system for the Unity inspector in which the user could select and save critical camera locations and orientations, and those locations and orientations would be merged into a path that the camera would follow throughout the

simulation. The hope was to make the layout of the custom inspector as user-friendly and intuitive as possible. Overall, this utility was implemented without any problems and the organization and functionality met all expectations as the system behaves as expected and improvements could always be made in the future if more features are added or if there is user-feedback about what users typically want.

There is a multitude of future work that can be completed regarding the best view orientation of 3D objects. Some of the feedback from the survey completed mentioned the lighting of the object affecting the view of the object, but in this research, the lighting was stationary and located directly in front of the object throughout the analysis so as to minimize the number of possible variables in the research. Lighting in the image could be investigated through a histogram of the colors of the pixels in the image. With this histogram, color gradients in the image or other patterns could be used to determine which lighting position favors the object in varying orientations. Another analysis that could be implemented as well could include metrics for analyzing inner features of the object, such as holes, grooves, etc., using cross sectional views for example. This histogram and inner feature research, along with the research in this work, could be implemented into a more formalized type of ML, in which a different type of machine learning could take place (e.g. supervised learning (*k*-NN algorithm is included in this), unsupervised learning, or reinforcement learning). It is even possible to implement some ML software such as TensorFlow into Unity using add-ons.

More future work can be done to improve the results shown to the user as well because for certain objects, the current results include views that are similar and could be mistaken for the same angle of the object. This occurs because of the symmetry of the object. For example, the tire is symmetrical on all planes in both the vertical and horizontal direction, so when rotating and taking pictures, some of the images would be the same. Duplicates could be eliminated through checking the current picture with any previous pictures taken. This is important because if more than one image (e.g. top 3 views or top 5 views) are presented to the user, they should not all be the same image but show varying images from which the user could select their preferred image. The process of taking the pictures takes up to a few minutes when a large number of parts need to be analyzed, so it would be reasonable to implement offscreen rendering to allow miniature generation as a background job.

Alternatively, the photobooth could be implemented as a service and it could fetch parts from the scene to a separate rendering pipeline. Currently, this is difficult as programmatic export functions are not ready for the geometry and materials.

Generally for the MOSIM project, because this work was only created and tested for the manufacturing of a pedal car, the next logical step would be to implement this work into more manufacturing simulations where the best view of parts need to be selected. For MOSIM, industrial use cases include passenger car manufacturing, bus manufacturing, truck manufacturing, wood processing, and pedestrian simulation (Bär, et al., 2020). Outside of MOSIM, this work of "best view selection" has been used in other fields such as 3D scanning and model recognition. In 3D scanning, the 'next best view' is used to find certain views that help the software recreate the model the best that it can. In a way, this work is similar in that it is necessary to scan the object at varying orientations and calculate certain properties to determine if the view is considered worthwhile. At the end of the day, the applications of 3D object analysis are broad and the ability to select the best view of a 3D object could be implemented into many types of software to present to the user a view that is both appealing and useful to the work that they are completing.

REFERENCES

Autodesk Help, 2016. *Polygonal Modeling.* [Online]
Available at: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/Maya/files/GUID-7941F97A-36E8-47FE-95D1-71412A3B3017-htm.html
[Accessed 11 Nov 2020].

Bär, T. et al., 2020. *WP6 Industrial Use Cases & Exploitable Results.* s.l.:s.n.

Cambridge University Press, 2020. *Assembly Line.* [Online]
Available at: https://dictionary.cambridge.org/dictionary/english/assembly-line
[Accessed 10 Nov 2020].

Falck, A.-C., Örtengren, R. & Rosenqvist, M., 2012. *Relationship Between Complexity in Manual Assembly Work, Ergonomics, and Assembly Quality.* Stockholm, Sweden, s.n., p. 91.

Gaisbauer, F., 2019. *The MOSIM project: combining digital human simulations - Unite Copenhagen 2019.* [Online]
Available at: https://www.slideshare.net/unity3d/the-mosim-project-combining-digital-human-simulations-unite-copenhagen-2019-182883909
[Accessed 10 Nov 2020].

Gaisbauer, F., Klodowski, A. & Antakli, A., 2020. *Demonstrator "Pedal Car".* s.l.:s.n.

Gaisbauer, F. et al., 2019. Proposing a Co-simulation Model for Coupling Heterogeneous Character Animation Systems. *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications : GRAPP*, Volume 1, pp. 65-76.

Haagndaaz, 2016. *Easy Curved Line Renderer [Free Utility].* [Online]
Available at: https://forum.unity.com/threads/easy-curved-line-renderer-free-utility.391219/
[Accessed 8 Sept 2020].

Haas, J. K., 2014. *A History of the Unity Game Engine.* Worcester(MA): s.n.

Halabisky, B., n.d. *Euclidean Distance In 'n'-Dimensional Space.* [Online]
Available at: https://hlab.stanford.edu/brian/euclidean_distance_in.html
[Accessed 29 Oct 2020].

Handel, C., 2020. *WP2 Status & Exploitable Results (MOSIM).* s.l.:s.n.

Locks, F. et al., 2018. Biomechanical Exposure of Industrial Workers – Influence of Automation. *International Journal of Industrial Ergonomics,* pp. 41-52.

Michigan Technological University, 2010. *Mesh Basics.* [Online]
Available at: https://pages.mtu.edu/~shene/COURSES/cs3621/SLIDES/Mesh.pdf
[Accessed 11 Nov 2020].

Microsoft, 2020. *C# Docs - Get Started, Tutorials, Reference. | Microsoft Docs.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/csharp/
[Accessed 10 Nov 2020].

Mitchell, T. M., 1997. *Machine Learning.* New York: McGraw-Hill.

Navlani, A., 2018. *KNN Classification using Scikit-learn.* [Online]
Available at: https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn
[Accessed 29 Oct 2020].

Ni, K. S. & Nguyen, T. Q., 2009. An Adaptable k-Nearest Neighbors Algorithm for MMSE Image Interpolation. *IEEE Transactions on Image Processing*, 18(9), pp. 1976-1987.

Unity Technologies, 2020. *Unity Platform.* [Online]
Available at: https://unity.com/products/unity-platform?_ga=2.247603655.1569157613.1599205949-1460250388.1599205949
[Accessed 10 Nov 2020].

Unity Technologies, 2020. *Unity Simulation.* [Online]
Available at: https://unity.com/products/unity-simulation
[Accessed 10 Nov 2020].

Unity Technologies, 2020. *Unity User Manual.* [Online]
Available at: https://docs.unity3d.com/Manual/index.html
[Accessed 10 Nov 2020].

Wang, Y., 2011. *Foundations of Intelligent Systems.* Shanghai, China, s.n., pp. 556-568.

Weber, A., 2016. *Mixed-Model Assembly Is Key to Profitability.* [Online]
Available at: https://www.assemblymag.com/articles/93387-mixed-model-assembly-is-key-to-profitability
[Accessed 10 Nov 2020].

Yang, K. & Jie, J., 2011. *The Designing of Training Simulation System Based on Unity 3D.* Nanchang, China, s.n., pp. 976-978.

APPENDICES

Appendix I: Training Data Points and Plots for *k*-NN Algorithm

Table 6: Training Data Points for the Best Views for all Metrics

Part Name	Projected Area	Surface Area	Center of Mass X	Center of Mass Y	Symmetry	Visible Edges	Mesh Triangles
Mainframe	0.044708	0.421521	-0.03506	0.190773	0.692124	0.014044	0.406043
	0.050586	0.437743	-0.00997	0.159245	0.579165	0.016215	0.410182
	0.048373	0.445741	0.014822	0.310936	0.6593	0.01475	0.40894
Right-rear Mudguard	0.086318	0.166629	0.017199	0.185241	0.391068	0.010673	0.266869
	0.083017	0.224285	0.008179	0.253984	0.255827	0.00981	0.334344
	0.0865	0.271931	-0.00816	0.277657	0.308379	0.009581	0.210766
Right screw	0.102788	0.166172	0.005532	0.598688	0.543495	0.007778	0.290323
	0.096645	0.161566	-0.00255	0.501108	0.933713	0.007656	0.282258
	0.102188	0.186832	0.024276	0.546098	0.850088	0.007773	0.33871
Right Steering Link Inside	0.106246	0.298976	0.034222	0.374776	0.483186	0.007577	0.278351
	0.106603	0.336982	-0.02209	0.375261	0.523486	0.007385	0.386598
	0.104702	0.315515	-0.02233	0.348609	0.631757	0.00715	0.329897
Hammer	0.06847	0.235714	-0.06552	0.37527	0.110126	0.008013	0.291667
	0.073047	0.557143	-0.02755	0.393355	0.373906	0.00926	0.541667
	0.070014	0.271429	-0.10888	0.362677	0.132973	0.008828	0.375
Ratchet	0.056157	0.101861	-0.00884	0.618535	0.449746	0.006573	0.065625
	0.054591	0.103293	-0.00108	0.595762	0.269253	0.006296	0.128125
	0.055891	0.09216	-0.10055	0.490051	0.322718	0.005172	0.059375

Table 7: Training Data Points for the Worst Views for all Metrics

Part Name	Projected Area	Surface Area	Center of Mass X	Center of Mass Y	Symmetry	Visible Edges	Mesh Triangles
Mainframe	0.024968	0.344227	-0.03489	0.152051	0.490722	0.008448	0.246275
	0.036614	0.409514	-0.00083	0.062642	0.737895	0.011314	0.380381
	0.017108	0.22898	-0.04788	0.356127	0.491822	0.006986	0.238411
Right-rear Mudguard	0.044898	0.164191	-0.0051	0.191795	0.692871	0.003539	0.100834
	0.084658	0.223902	-0.00534	0.362953	0.771963	0.007298	0.18044
	0.140309	0.382485	0.028872	0.165564	0.869586	0.014298	0.285064
Right screw	0.054859	0.037852	0.002675	0.177493	0.894934	0.005763	0.096774
	0.054845	0.192563	0.00449	0.171425	0.927878	0.004555	0.298387
	0.070323	0.380767	-0.00521	0.226467	0.339293	0.005967	0.362903
Right Steering Link Inside	0.06525	0.205563	0.003303	0.200841	0.99785	0.002786	0.185567
	0.065267	0.030491	-0.00099	0.192614	0.922702	0.006506	0.041237
	0.102633	0.277247	0.003553	0.377306	0.703317	0.006674	0.265464
Hammer	0.02693	0.121429	0.001269	0.181284	0.934068	0.003414	0.166667
	0.02693	0.107143	0.001269	0.219312	1	0.002626	0.083333
	0.038325	0.196429	0.024972	0.175	0.293694	0.00487	0.25
Ratchet	0.013468	0.001432	0.001308	0.086938	0.962568	0.001975	0.0625
	0.019009	0.118249	-0.00142	0.126909	0.39757	0.003411	0.115625
	0.020056	0.093592	-0.00125	0.11695	0.252322	0.003328	0.121875

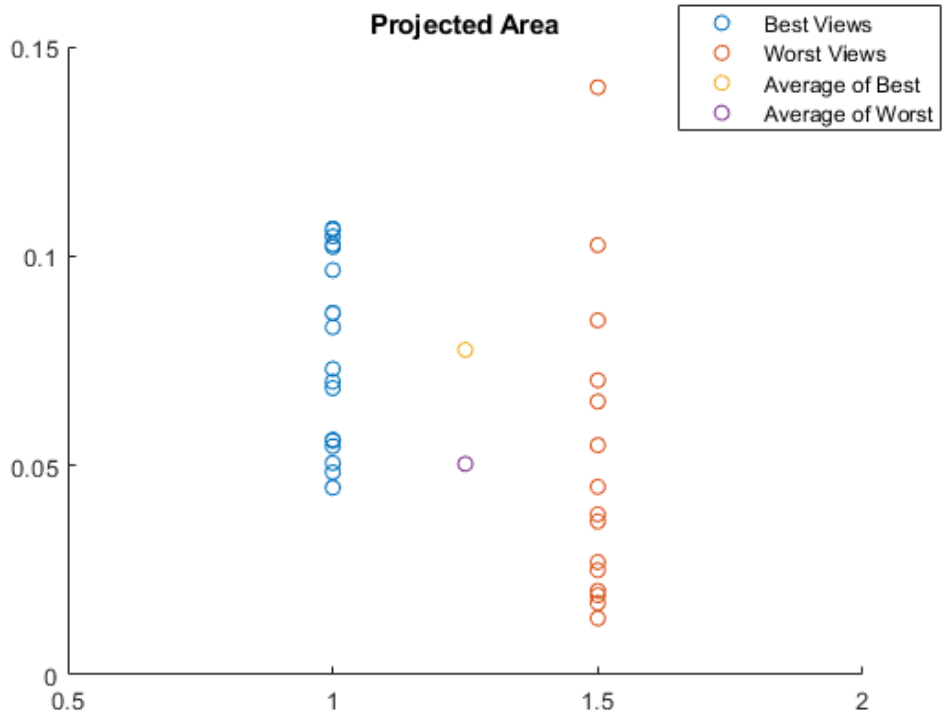


Figure 101: Projected Area Training Data Points (Best and Worst)

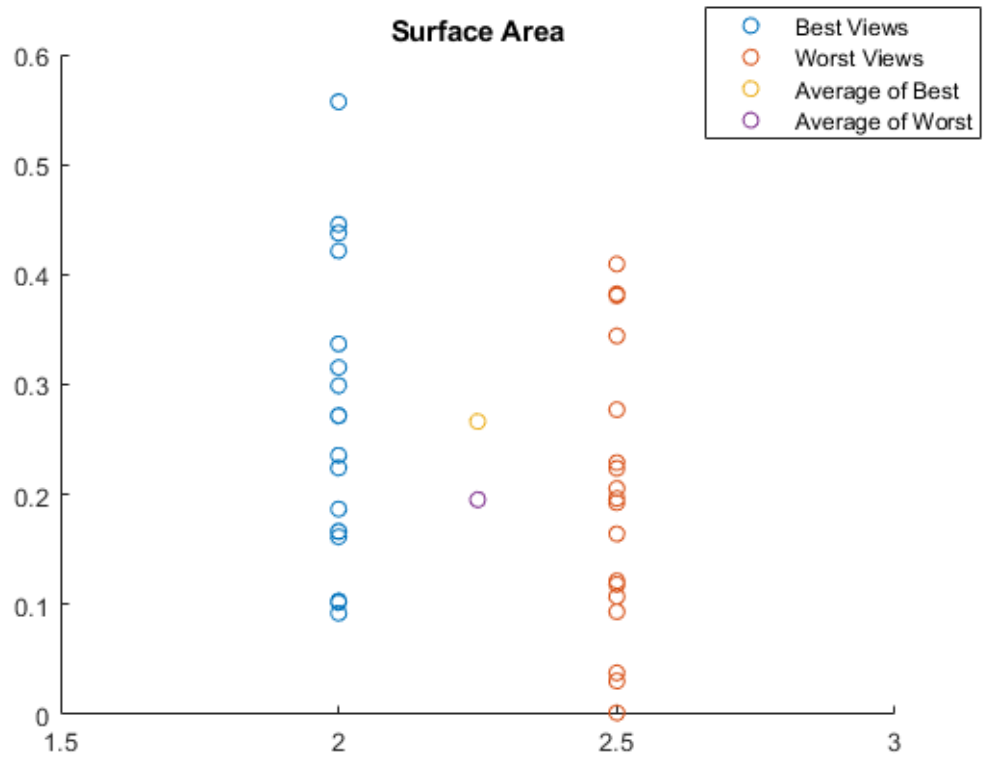


Figure 102: Surface Area Training Data Points (Best and Worst)

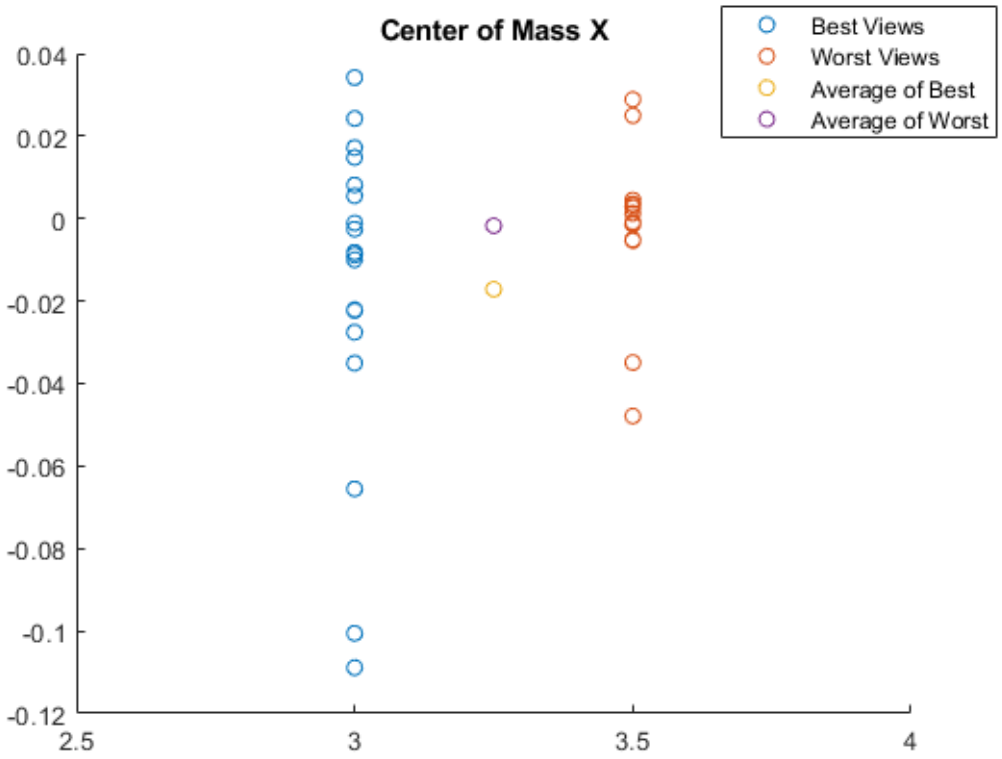


Figure 103: Center of Mass X Training Data Points (Best and Worst)

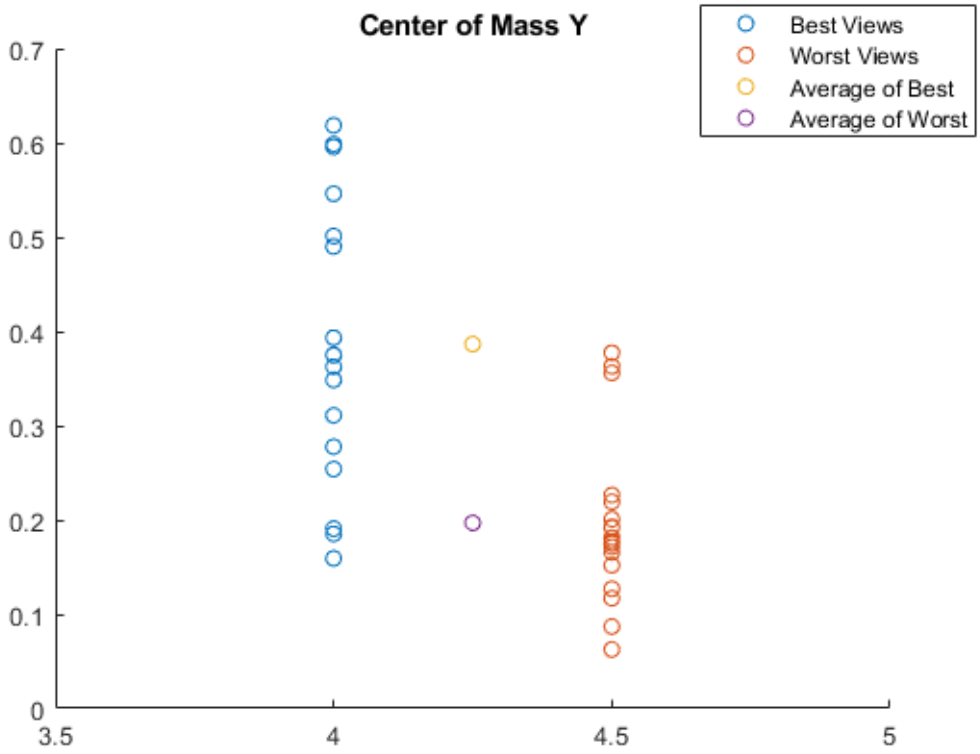


Figure 104: Center of Mass Y Training Data Points (Best and Worst)

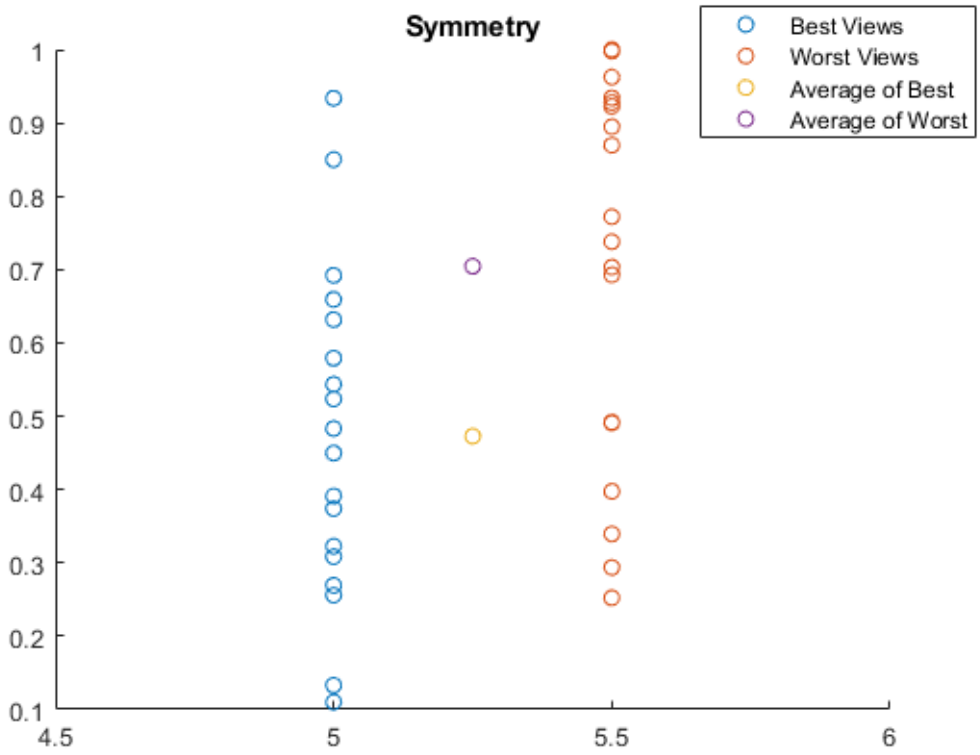


Figure 105: Symmetry Training Data Points (Best and Worst)

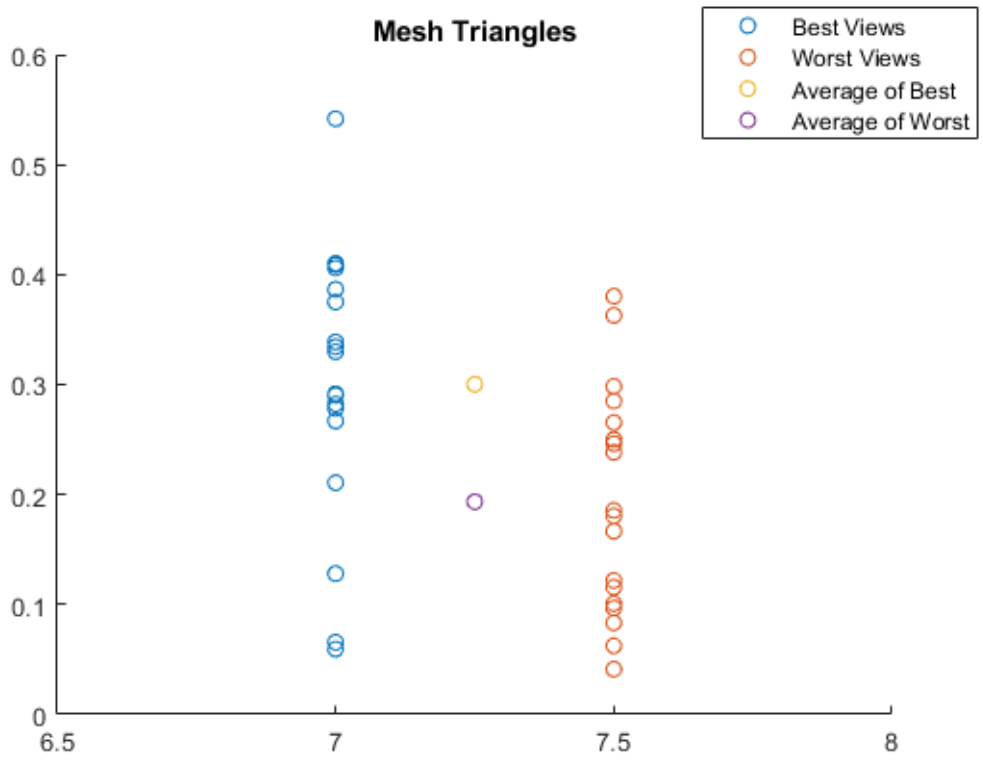


Figure 106: Mesh Triangles Area Training Data Points (Best and Worst)

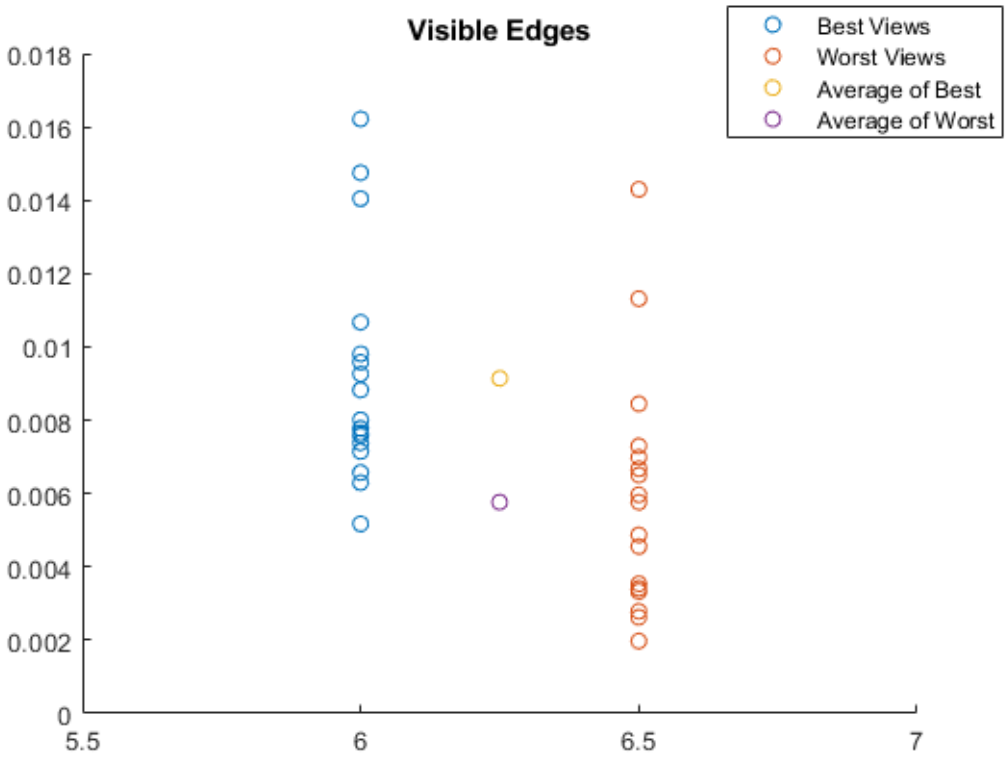


Figure 107: Visible Edges Training Data Points (Best and Worst)

Appendix II: Survey of the Best Orientation between Procedures

Best Orientation of Object

For this survey, simply select the view that you think best describes that object. (Should take 30-60 seconds maximum to respond.) Thanks!

*Pakollinen

Select the view that you think best describes this frame: *

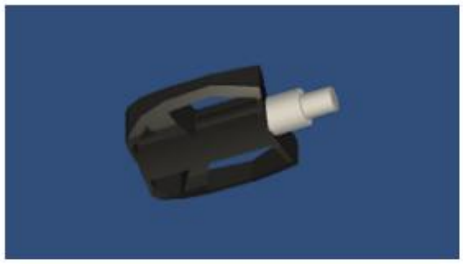
Orientation 1

Orientation 2

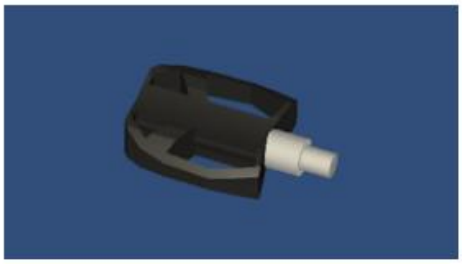
Orientation 3

Orientation 4

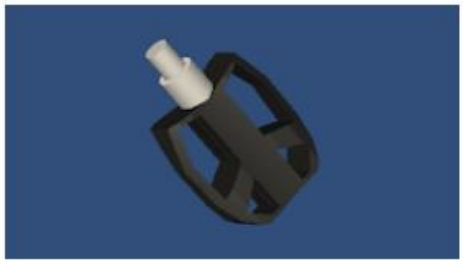
Select the view that you think best describes this pedal: *



Orientation 1



Orientation 2



Orientation 3

Orientation 4

Select the view that you think best describes this steering wheel: *

Orientation 1

Orientation 2

Orientation 3

Orientation 4

Select the view that you think best describes this tire: *

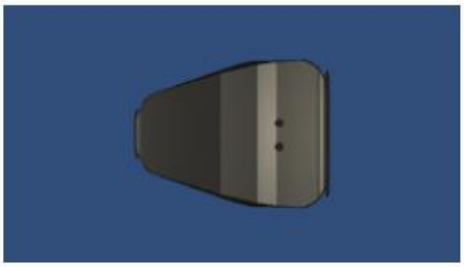
Orientation 1

Orientation 2

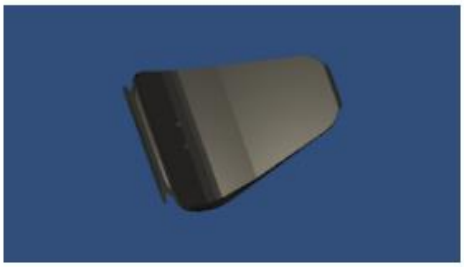
Orientation 3

Orientation 4

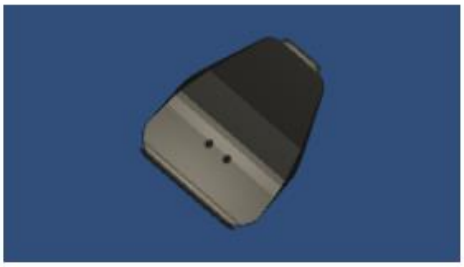
Select the view that you think best describes this seat: *



Orientation 1



Orientation 2



Orientation 3

Orientation 4

Select the view that you think best describes this fixture: *

Orientation 1

Orientation 2

Orientation 3

Orientation 4

(Optional) What was the main factor for your choices above?

Your own answer

send