Final Design Document

G-02 CEM-VR

Aidan Brightman, Chad Greenspan, Garth Simpson, Jacob Peach, Samuel Watts

Sponsors: Dr. Amy Giroux, Dr. Brook Miller

Table of Contents

Executive Summary

This project has the goal of continuing the creation of a virtual reality simulation of the St. Augustine National Cemetery, an endeavor started by Dr. Amy Giroux, and will also be serving as a template for similar projects in the future for other national cemeteries.

The top objectives of this team are to:

- ❖ Create an automated system for texturing the most common headstone type
- ❖ Improve the running time performance
- ❖ Investigate and fix pre-existing errors
- ❖ Enhance the visuals through additional models and designs
- ❖ Ensure development tools are easily reusable for future similar development projects

Each objective will be elaborated upon in future sections.

This project has inherited a codebase through an established github code repository that has been forked by this team and built upon using the git version control. The technical aspects of it can be broken up into the different services being used within the repository, those being the unfinished automated texturing tool, the Unity project the simulation runs on, a database tool created by Dr. Giroux, and new unit testing features provided by github to work with the Unity system.

Introduction

Societal Impacts

The St. Augustine National Cemetery Virtual Reality experience has the cultural purpose of honoring the lives of those who had fought in for their country, served for their country, and died for their country. It also serves as a digital conservation effort to permanently preserve the memories of not just the individuals, but the architecture, symbolism, and design of the historic site. Having the capabilities to tour the national cemetery allows the experience to be brought

all around the country to individuals who might not see it, and for people who might not have the physical means of navigating the location.

In addition, this project will serve as a blueprint for future projects that aim to create virtual reality simulations of other cemeteries around the nation as well. The more modular and reusable the code is for this project, the easier it will be to preserve more historical cemeteries throughout the nation.

Individual Significance

Aidan Brightman

My name is Aidan Brightman and I am the project manager, product owner, and software engineer for the St. Augustine National Cemetery Virtual Reality experience led by Dr. Amy Giroux and Dr. Brook Miller. For work I am a part-time Junior Software Engineer at Maxar Intelligence, working with lots of devops tools as well as doing some full stack development. I was drawn to this project by the prospect of learning more of the Unity Engine, as I have been developing a hobby interest in game design. To me, my love of programming comes from finding ways to apply its logical structuring and system design in ways to construct creative mediums. This is something I believe is culminated in projects like game design, as well as artistic creative endeavors like this virtual reality tour of the historic St. Augustine cemetery. I already have experience from the course AI for Video Games, as well as from personal projects I have worked on, but I haven't worked on a 3D scene, nor have I worked with heavy optimizations under hardware constraints. In doing this project I hope to learn those skills for future application on personal endeavors.

As I have experience with Scrum at my job, I was suggested to be the project manager as well as the product owner. This position of leadership has motivated

me greatly to do the best that I can on it. Knowing that the actions of the group will reflect on my character motivates me to keep workflow organized, lines of communication open, and morale positive. When working in a group, I have been drawn to leadership positions as I feel that communication and division of labor have been strong suits of mine. In addition to this, I have been learning more skills related to systems design at my job, which I feel has taught me organizational and resource allocation skills as a result. Learning each of my group mates strengths, experiences, and limitations have resulted in a group that I believe is having its attention divided in a greatly effective way.

For this project, I am working with Samuel Watts to optimize the Unity project for the Meta Quest 2 VR headset and for fixing bugs left in or caused by previous groups. In addition to fixing the bugs, I am also taking to implementing scripts and procedures that will hopefully prevent errors from repeating in the future.

Chad Greenspan

My name is Chad Greenspan and I am a member of the third group to participate in the construction of the St. Augustine National Cemetery Virtual Reality experience. During the ranking/picking process for senior design proposals, I was in the hospital–thus leaving my project up to chance. To my delight, I was selected for a project which deals heavily in 3D architecture, computer graphics and modeling. I have a significant background in CAD-based modeling for production of physical products, but my initial understanding of modeling for virtual consumption was only circumstantial–due to the overlap of modeling softwares. However, I have experience in blue-light scanning of real-life objects and then converting them into machine-readable 3D objects. In one instance, I used a Revopoint Pop 3 to scan a 3”x3” object with over 600 features,

combined multiple point clouds from multiple scans and then converted those point clouds into a polygonal mesh–which I then decimated to reduce load while accessing the file and to reduce file size. This polygonal mesh was then exported as a .STL, which is readable within engines such as Blender. Additionally, I have real-life engineering experience in the DOD industry, albeit not in any computer science disciplines. Thus, this project gives me tangible experience for future prospects that apply to my field of study.

It did not take much discussion with my team to come to the conclusion that I should be focusing on the modeling aspect of this project. Initially, we were under the assumption that some of the unique headstones were not modeled and this would need to be done. This led to a debate–mainly internal, due to my extensive experience in the latter–whether this modeling shall be done in traditional game engine modeling softwares such as Blender, or whether it shall be done in CAD. This is further discussed in the “Additional Modeling and Texturing” section of this document.

Additionally, I will be assisting Garth in the texturing aspect. Some texturing is unable to be automatically applied to certain models due to lighting, perspective, or focus issues present within the photographs provided. Due to the distance of St. Augustine from our residences, it is not feasible for us to retake pictures for texturing purposes if one cannot be effectively applied. Thus, manual texturing will be a requirement for ease of manufacturing this end-product.

Garth Simpson

My name is Garth Simpson and I have been assigned to Group 2 CEM to work on a VR walkthrough of the St. Augustine Cemetery. The primary reason I rated this proposal high on my list is because I wanted more experience with python and game development, particularly for VR games. In my upper electives

for Computer Science, I have studied computer graphics, computer vision, and machine learning so I felt confident I would be able to fulfill a role in this project. In the proposal, I noted the automated texturing for the cemetery headstones used AI to extract faces of the headstones from an array of images. Computer graphics taught me the 3D OpenGL rendering pipeline detailing how to create a simple rendering engine with TWGL and so I was able to learn how vertices can influence the performance of a 3D render. I have done AI training before for object identification, and in computer vision I made code to turn angled pictures of documents into flattened pictures using canny edge detection and matrix math. Currently, I am dual majoring in Experiment Animation specializing in 3D Art and 3D animation. My goal is to become a 3D artist for video games and simulations. Being a part of a VR project would give me more experience with the VR game development pipeline and of VR performance limitations.

The role I took was for fixing the computer vision programming of the project. I spent some time looking through the code of the previous teams' and after talking to Dr.Giroux I believe the project needs mainly optimization. The second team was unable to make the headstone texturing process automated according to Dr.Giroux which means the python code has been largely untouched since the first team. Part of the python code is run separately from the unity project likely due to python being a slow and performance intensive language. The thought of having all automated texturing occur at runtime did cross my mind, but considering the project has performance issues already, this could make the game unstable. Additionally, running an AI model everytime the application starts could lead to inconsistent results. My goal is to optimize the python code so it is applicable to different tombstones, not just the tombstones of the St. Augustine cemetery.

Jacob Peach

My name is Jacob Peach, and I'm a participant in the CEM-VR project, which aims to translate the real life space of the St. Augustine National Cemetery to a VR experience on the Oculus Quest 2. What drove me to this project is my experience in my Computer Vision and Computer Graphics courses I took here at UCF, which I found to be fascinating. I thought for a long time about how the low level, matrix based concepts in those classes were translated into real graphics with performant, repeatable effects. I have little experience in game development, and a lackluster set of experience within coding as a whole, but seeing the effects of so many small parts that comprise a greater whole is a motivating factor for me, and there's few places that exhibit this better than computer graphics. My motivations in the CEM-VR project lie along those lines. The issues plaguing the current build are numerous, but certain aspects - namely the texture issues - instantly caught my eye and gave me ideas about what could change about it.

As far as my workload is concerned, my role is to work closely with Garth to identify, document, and fix the issues in the CEM-VR automatic texturing system that are affecting the look of the headstones. This system is meant to take the image provided for a certain headstone, and map it to the correct face of the few hundred headstones in the cemetery. The first part of this, identification, will involve learning the ins and outs of the previous team's texture tool completely, the value of which being gaining and understanding of its functions. With this, the next step, documentation, becomes easier. This documentation, which ideally will be a full explanation of the system at a high level, is both important for the later required writings within Senior Design 1, and for the future goal set forth by Dr. Giroux, of having something Generalizable. This also ties into the fix for these issues, and the way that these issues are fixed. If changes to the existing tool

beyond the texturing fix could serve the goal of this tool's future use, then those changes, in my opinion, are justifiable.

Samuel Watts

My name is Samuel Watts, and my motivation for choosing the St. Augustine Cemetery in VR comes down to an interest in modeling and simulation work, C# programming, and unit testing. Unity is brand new to me, however I do work as a CWEP intern where 50% of my time is spent on stories with the Modeling & Simulation team. This type of work is very interesting to me, and I want to learn more about this particular field of software development. M&S (Modeling and Simulation) is my main focus for careers post-grad. CEM-VR gives me another opportunity to focus on realistic environment simulations in another language. C# is a language I have not had hands-on experience with yet. This project will give me exposure in game/M&S scripting for handling user interactions as well as object interactions in the world environment. Lastly, I wanted to grow my unit testing skills and CEM-VR seemed like a great fit. As a CWEP, the other 50% of my time is with the Devops team, creating, editing, and bug fixing of metric and memory based unit testing. This capstone project will allow me to learn more about unit-testing from a less-numerical side. I will be creating CI/CD pipelines in Github to validate physical collision detection(EX: flowers should rest on top of gravestone, not fall through), query tool functionality (broken previously due to no unit testing), and object behavior.

My ideas largely focus around bug fixing and optimizing current operations. Previous work had broken vital functions of the VR program due to a lack of central unit testing on commits. On each commit for every branch, a Github pipeline will kick off a series of unit tests. This will create a system for quickly

determining if a team member's updates have inadvertently broken another section of the program, while also giving members a consistent testing environment. This will avoid “It worked when I tested it” scenario’s if all tests are visible in Github and all tests use the same values for consistency. Certain bug fixes will fall into my area of focus as well. The query tool for accessing and modifying the program's database is currently inoperable. This functionality will need to be restored before accurate unit tests can be implemented. I enjoy using the debugger to go line by line through code and see how it is behaving. Another bug fix in my realm will be editing the colliders to max sure real-world physics are restored. This will also apply to other bugs that may pop up when working on the program. I will also be working alongside Joe (Aidan) to focus on program optimization and efficiency. The general idea is to focus on reducing unnecessary resources to maintain smooth performance.

Ethical Responsibilities

As we are creating a simulation for headstones in a cemetery, many of them also belonging to individuals who served for the military, it is important we portray the location with the utmost sincerity and respect. The efforts to create an automated texturing tool for the headstones has the ethical responsibility of representing the headstones in the most realistic way. Absurd skewing and warping of the imagery can be seen as not only inaccurate to the real life design, but disrespectful to the individual for portraying their final resting place in a manner that could be considered “goofy”.

In addition to this, we have the ethical responsibility of doing what we can to ensure users maintain good health while in the simulation. Virtual reality plays with the body’s perceptive tools, and poor quality simulations can cause

discomfort in users. Poor visual quality can lead to eye straining and headaches, while unresponsive or delayed movement can cause nausea for users. It is important that the product that we create is made with a standard of care that prevents users from experiencing this discomfort.

Project Characterization

Concept of Operations

The St. Augustine National Cemetery Virtual Reality experience is intended to be used to allow people to travel around a simulated version of the cemetery. Users will be able to interact with the cemetery in ways such as:

- Move around the cemetery
- Querying those buried through different criteria
    - This will highlight headstones dependent on filters.
- Press a button on headstones to collect additional info on the deceased
- Put coins and roses on the headstones
- Press a button on the base of the flagpole to play either
    - The Star Spangled Banner
    - Taps

The means users will be able to access this simulation will be through the non-tethered Meta Quest 2.

Technical Overview

Here is a current list of required technical tools that will be used in the construction of this project:

- Unity Engine
  - Version 2021.3.8f1
- Visual Studio
  - IDE with built in Unity capabilities
- Meta Quest 2 Virtual Reality Headset (Untethered)
  - Hardware the end result must be able to run on.
  - Quality assurance will be measured relative to this hardware
- Google Collab
  - Primarily used to test for problems in the previous code ran on Google Collab
  - Runs the projects current AI model, Detectron2
- Github
  - Stores the code and provides useful UI for group development
  - Github actions will be used to kick off CI/CD Pipelines
- Blender

- ➢ Used to generate new models at the request of the sponsor
- ➢ Used to optimize polygonal mesh of previously generated models

❖ Substance Painter

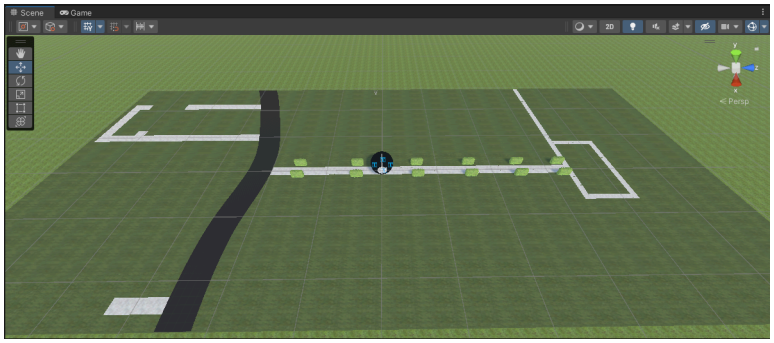
- ➢ Used to create manual textures for the houses and Obelisks

Running Simulation Via Tethered Headset

Steps to follow in order to get the simulation running via a tethered oculus quest is as follows:

- ❖ Get project running in Unity
  - ➢ Install Unity Hub
  - ➢ Pull repository from Github
  - ➢ Install Unity Editor version 2021.3.8f1
  - ➢ Run project
- ❖ Download Meta Quest App
  - ➢ https://www.meta.com/gb/quest/setup/?utm_source=www.meta.com&utm_medium=dollyredirect
- ❖ Complete account set up
- ❖ Link via cable the PC and the headset
- ❖ Set OpenXR Runtime to “active” in oculus app settings
- ❖ Navigate to LoadScene scene within the Unity Editor

Upon playing the scene, the user should be able to explore the scene through the VR Headset. It is worth noting that the scene before runtime will only consist of the XR tools, some hedges, and the basic layout of the ground. Once the scene has been played, it will be populated with the contents of the CemeteryAssetsFull scene.



[1] LoadScene before runtime

[2] LoadScene during runtime

VR controls for the scene are as follows:

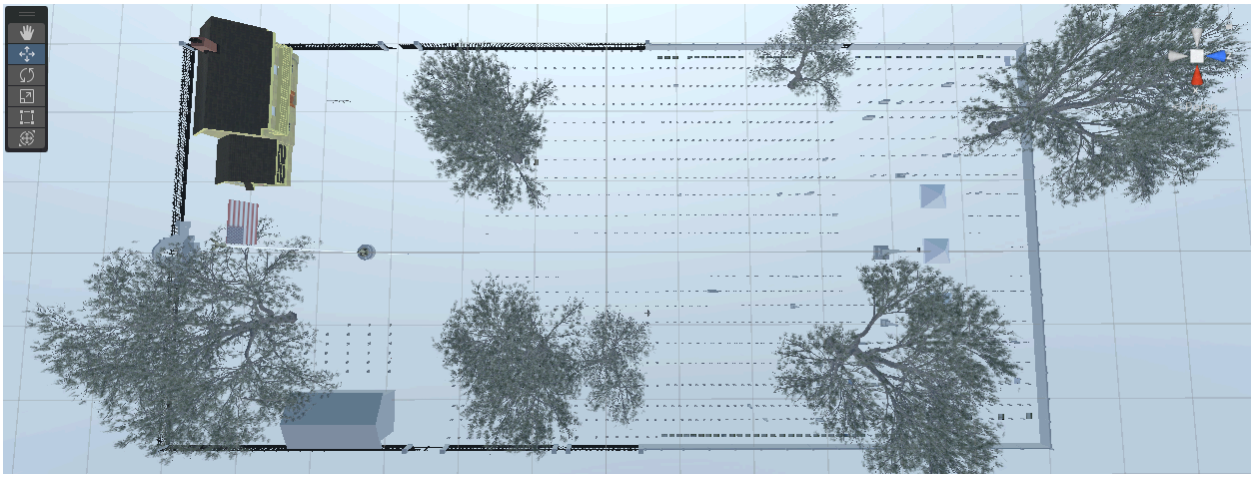
- Movement: Left Controller Analog Stick
  - Hold to send out teleport location, release to teleport
- Query Screen: A Button
- Select: Back Triggers

- ❖ Grab: Side Triggers

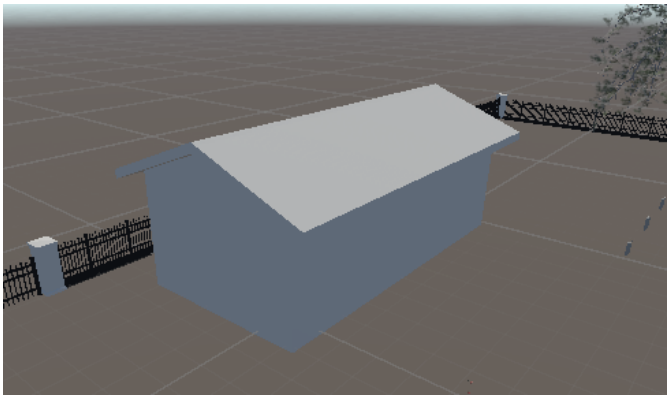
Miscellaneous Tasking

Within the Unity project, the CemeteryAssetsFull Scene is the main scene for gameplay. Within the scene there are a lot of minor adjustments that need to be implemented for the simulation to be more accurate.

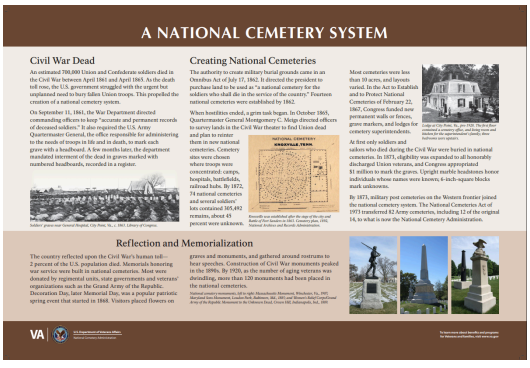
1. The east section of headstones, as well as the small group near the utility building in the northwest side of the cemetery, need to be adjusted to face eastward.
2. The utility building in the northwest side of the scene needs to be textured.
3. The Gettysburg Address plaque near the southern edge needs to be made larger.
4. An introduction plate needs to be updated to have a better texture supplied by the NCA.
5. There need to be additional headstones located sporadically throughout the scene.
6. Headstones need to be textured.
7. Some unique gravestones need to be modeled, textured, and put into the scene.
8. The set around the cemetery needs to be decorated.

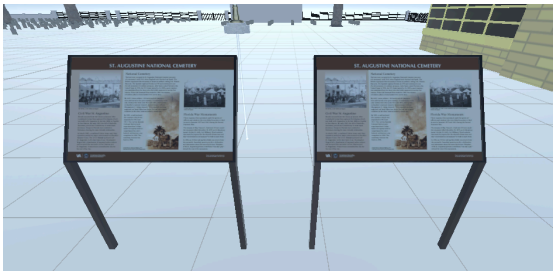


[3] Top down view of CemeteryAssetsFull Scene in Unity Project (top is east)

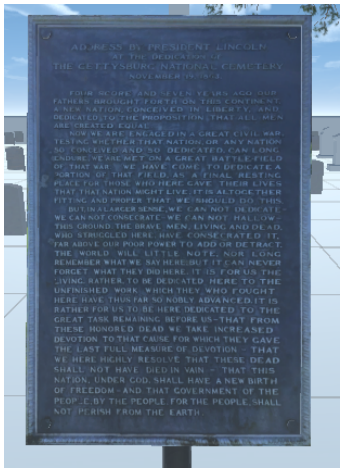


[4] Untextured utility building

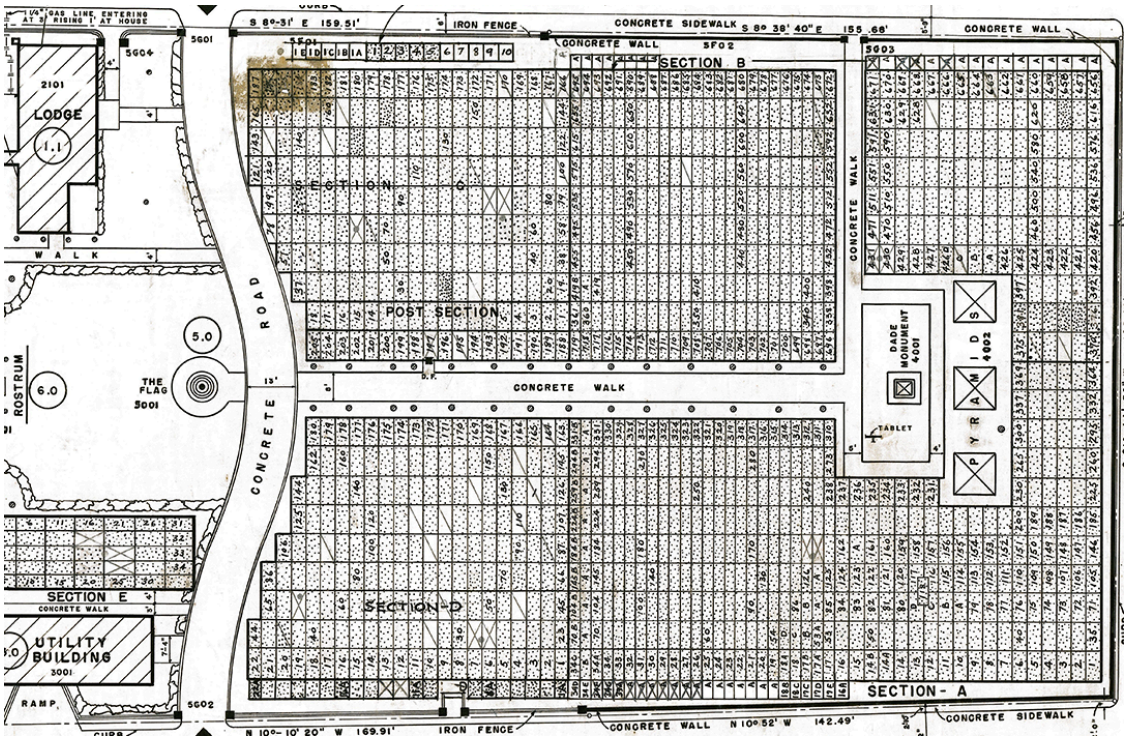




[5] Information plaques and the updated picture to change the left board



[6] Screenshot of the Gettysburg Address that needs to be scaled



[7] Official layout of the St. Augustine Cemetery from 1957, being used to fact check headstone count + placement, provided by Dr. Giroux

Bug & Error Fixes

Known bugs within the Unity program the team has been tasked with fixing include:

- An unknown issue where the query tool for accessing the project's database values not functioning properly
    - This is elaborated further within the “Query Tool Investigation” section.
- Collision bugs associated with objects falling or passing through solid objects
- Submission paths for buttons that play music and more to be broken
- Textures not loading from a certain platform.
- A selection of headstones have been turned to face an incorrect direction.

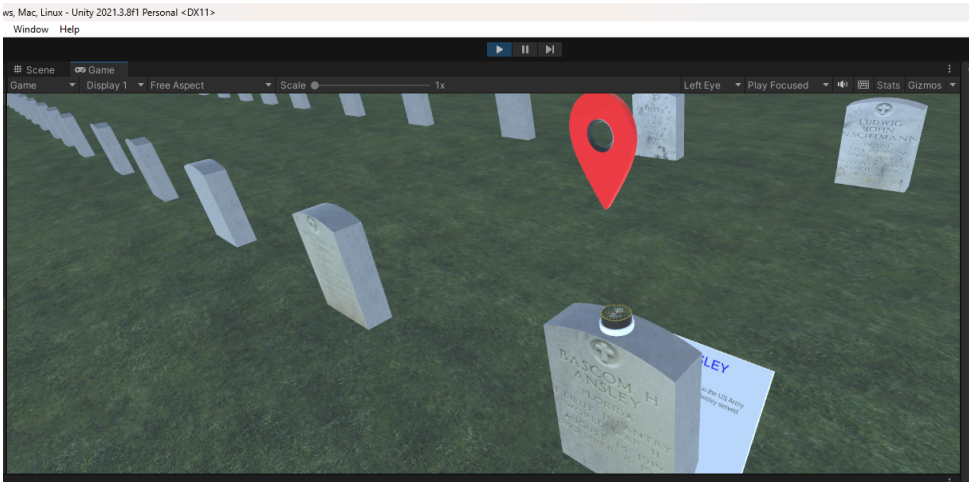
The issue of textures not loading from a certain platform has been addressed through eliminating the level of detail which culls the model. The level of detail that will remain contains only a small dozen of triangles, which should not have a significant impact on the performance while enabling the player to stay immersed within the simulation.

These known bugs will be noted as bugs in the Jira backlog. Any other bug discovered through experimenting with the program, or created through team commits, will be noted as a bug in a Jira ticket. Bugs will be added to the following sprint of discovery, unless the team decides the bug is impactful enough to warrant a hotfix in the current sprint. These cases will be decided by the whole team.

To ensure that errors such as these are minimized by future groups, a system was implemented to highlight objects that do not follow a directional standard put in place by a prefab object. This will allow quick and easy checking for directional debugging. This is elaborated upon further in the Unity Implementations section. For other issues, such as the querying issues, troubleshooting procedures will be implemented to locate errors and streamline the correction process.

Query Tool Investigation

Within the project, there is the capability for users to query an SQL database which contains information pertaining to the individuals the headstones belong to. Users may query their branch of service, conflict, rank, unit, state, award, emblem of belief, and section. When the headstones are queried, it will showcase a red marker above headstones that match the established filters, and will show a button which can be used to show a screen that has the information within it. Users may adjust various other settings within different tabs, such as setting the environment to be night or day, rainy or sunny, and playing the national anthem or TAPS.



[8] A demonstration of a headstone matching a query

From the last team, there has been an inheritance of the query tool now having bugs attached to it. Initially the impression was that the query tool created by Dr. Giroux has been altered and the communication with the database has been broken. After further investigation, the issues regarding the query tool seem to lie within the menu itself. The basic search tab can be queried with the information provided, however once it has run one query, the drop down menus that are used to fill in the information are no longer able to be edited. Clearing the fields also does not do anything, and instead all menu interaction with that page becomes inconsistent and futile to work with. The other tabs work perfectly okay, as the environment tab still allows you to make all of the modifications, granted with some difficulty regarding the controller inputs.

Unity Implementations

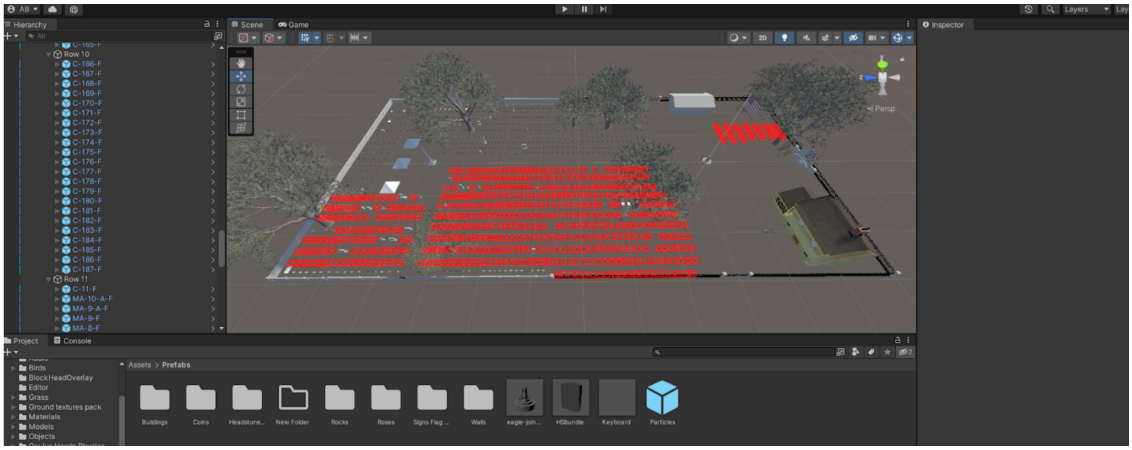
Unity Editor Scripts

Editor scripts are scripts within Unity projects that exist within the UnityEditor namespace and must be placed inside the Assets/Editor/ folder within the project. These scripts allow users to extend the functions of the editor itself, instead of affecting anything that happens during runtime. Through the utilization of editor scripts, users are able to create custom tools to automate menial and repetitive tasks and generally improve the workflow efficiency of their project. Since these scripts run within the Unity Engine editor, they can be used to make the creation progress more streamlined for debugging or testing.

An important feature of editor scripts is the ability to create menus within the editor using the MenuItem attribute within the script. When applied to a static function, this attribute adds a new command to Unity's toolbar, enabling easy access to the script's functionality from the editor's menu system. This allows users to perform specific actions within the GUI of the project, instead of operating within the debug log or reliant on runtime factors. Menu items can be used to toggle on and off debugging tools, reset game objects, or adjust project settings without needing to dig through various menus. They can also be used to create gizmos, windows tools, and inspector tools for various purposes to interact with the project.

One particular use of the editor scripts present in this project is being able to detect any object of a particular prefab within the scene and highlight them in the event of their y rotation being different than 0 (this is as we have set the 0 rotation to be facing eastward).

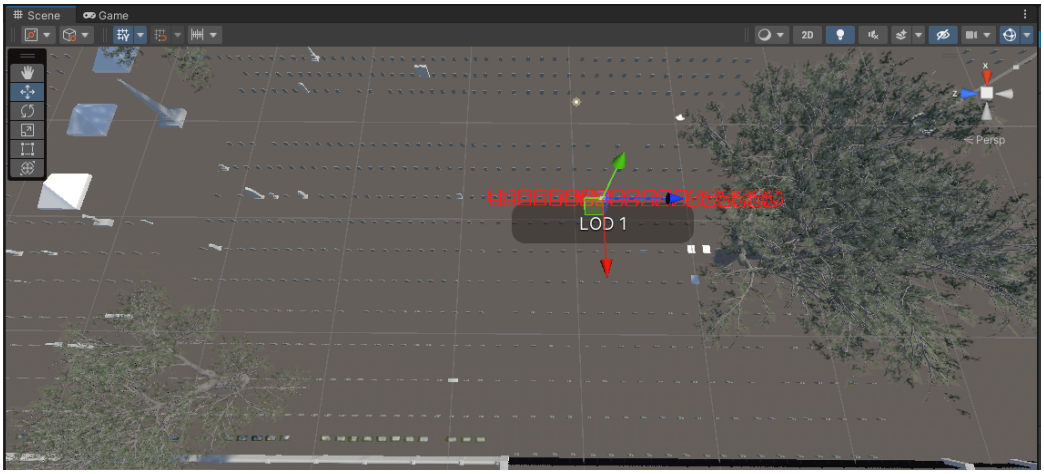
To accomplish this, the editor script has a readonly string that carries the path to the prefab within the file as well as a GameObject variable for the prefab. Whenever the Scene calls OnGUI the script will then check objects of the specified prefab and check the y rotation. If the rotation is anything but 0, then a red wire cube is drawn around the object. To toggle this on and off, a Menu Item is created to flip a boolean that will determine if the drawing happens.



[9] Example of highlighted headstones

During the implementation of this, errors have arisen. The first notable issue was a performance impact within the editor as a result of rendering the initial check. This issue was the reason the capability to turn on and off the boxes was implemented, as it allows the user a quick solution to reduce the load of the script. An issue that has persisted is a bug that prevents the script from running on the first load of the project. When a user opens the Unity editor, an error will be thrown claiming the non-existence of the HSbundle.prefab. Upon further inspection, the user can find that the HSbundle.prefab does in fact exist within the location it claims not to. In order to get the script running, the user must reattach the script to Unity through VSCode. Efforts to fix this problem are scheduled at the time of writing this.

Another issue that has persisted is a curious case of a group of headstones in a row being highlighted, despite their y rotation being 0. Even more bemusing is the fact that the row subset will no longer be eliminated if their rotation is turned to be 180 degrees. The check for the program relies on the objects Euler angle's y component, and will only highlight the object if the component is not set to be 0. Copying and pasting one of the incorrectly highlighted objects results in a headstone that behaves correctly, leading to more confusion.



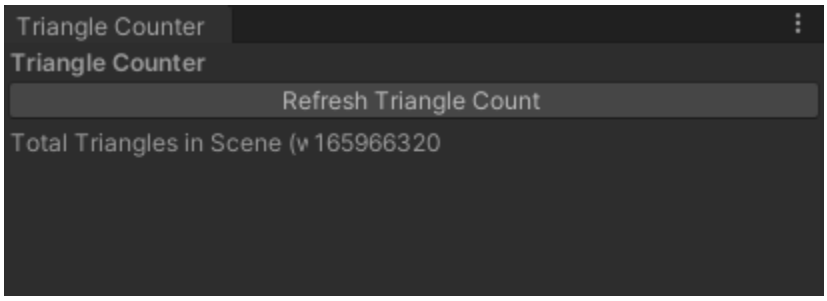
[10] Highlighted are turned to have a rotation of 0 and are still highlighted

Some theories on the cause of this problem have been either a technical error related to how the object is designed within its blender model file. A potential mistake with the set direction could cause the understanding of this angle to be misinterpreted. This is an unlikely cause, as if it were the case, it would likely be reflected in more than just this particular group of headstones. There is speculation of the error being caused by the relationship Unity has between euler angles and quaternion angles. Within the Unity documentation for Transform.eulerAngles, it warns that the engine does not utilize Euler measurements and instead stores the information as quaternion angles. It also informs that, because euler angles can be represented in multiple ways by quaternion angles, results can sometimes vary when calling for a euler angle. With this known, it is possible that the quaternion angles are being registered differently for this particular set of headstones. To solve this, specifying checks on the quaternion angles of the objects instead of euler angles may be necessary for correction.

In addition to highlighting the objects, there would be use for an editor script that enables the developer to get a live counter of the amount of triangles the camera is currently rendering. This allows whoever is working in the scene to

not only be able to recognize the load intensity, but also be able to experiment with how intense specific locations and areas will perform without the need for the headset in testing.

The editor script calculates the total triangles that are detected by the scene view camera by finding all the mesh filters currently visible by the camera, then adds the triangle count of that mesh to the total. It then does the same for skinned meshes, which is only utilized by the birds in the scene so its effects are arguably negligible within this scene. It then checks the level of detail of the objects and adds those into the triangle count. The script also has functions to create the refresh triangle count, which can be used to ensure the count is up to date, as well as the code to make the window accessible through the Unity window tab.



[11] An example of the window used for calculating triangles

This script has granted greater insight into some underlying problems within the project that have been impacting the performance. The Oculus Quest 2 suggests a range of 750k to 1 million triangles counted per frame. The triangle counter has shown amounts nearing three million within the scene from certain angles. Within the running time execution of the LoadScene scene, which contains more models and is the final product, there have been readings of 165 million triangles. This exceeds even the recommended triangle count of the Oculus Quest 3 by approximately 164 million. With efforts on the team moving towards ways of

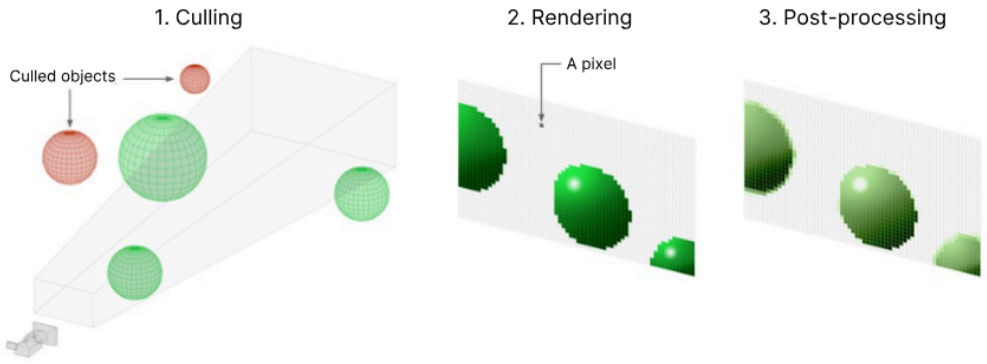
decimating the models, in addition to substituting the abysmal tree models within the scene, the triangle counter can act as a convenient way to check the counts for the user to verify we fall within the technological needs for the hardware.

Rendering

A large cause in poor performance for the simulation has to do with a high amount of computational power being used on rendering everything on the screen. Performance issues can be mitigated through bringing down the number of calculations needed for the GPU to perform. An overabundance of triangles needing to be rendered is a common way to load the GPU with calculations to slow down the program.

Rendering Pipeline

Rendering pipelines are a vastly important procedure which can determine both the graphical capabilities and the hardware limitations. A rendering pipeline is the procedure the engine takes to generate the 2D image frame from dynamic 3D scenes. The process begins with taking the camera into account to generate what exactly is seen to decide what objects are going to be rendered, frustum and occlusion culling which is elaborated on later in this section. After this, the geometry of each of the models has their vertices transformed to fit the perspective of the camera in the scene. A projection matrix is then finally applied to generate a 2D projection of the 3D perspective. From here, texturing and lighting is calculated to determine the color that each pixel should be before entering the post processing stage, where additional effects are processed on the visuals.



[12] Highly abstracted render pipeline example from Unity's Manual

The calculations that the computer must do during each of these steps can be intensive on the GPU when high amounts of triangles are being rendered at once. This is because large amounts of matrix calculations need to be applied to the triangles from the vertex shader. Matrix transformations require an incredibly large volume of simple calculations. Parallel processing is used in most modern day GPUs to do many of these simple calculations at the same time as one another, making it go by faster. That being said, the hardware only has a limited amount of shader cores to handle this processing. Once the triangle count rises to the millions, the number of calculations that need to be processed will outscale the capabilities of the GPU, causing delays and framerate drops for the program.

Within Unity, there are many ways to tailor the rendering pipeline to what the project requires. Unity, by default, has a standard rendering pipeline which is sufficient for most projects made with it. Alternatively to this, there is what is called the "High Definition Render Pipeline" which allows for higher end graphics and more capabilities when it comes to graphical performance. It supports advanced techniques for more fine tuned lighting, such as volumetric lighting and ray traced ambient occlusion. This makes it a wonderful rendering pipeline to use when graphics are a high priority. The caveat of this is that it is highly intensive

computationally and is only for new gen consoles or high end PCs. The HDPR is recommended to only be used in situations where the hardware is expected to meet high standards. This means that it is likely not a good option for this particular project, as the Oculus Quest 2 would likely struggle with the strain of the intensive calculations needed.

Unity also offers the “Universal Rendering Pipeline” for developers. While this has less graphical capabilities, it is much more flexible with devices and has a good blend of graphic support and compatibility with lower end hardware. It has the added benefit of a stated focus on working specifically with untethered virtual reality machines. For graphical capabilities, it has a VFX graph tool which allows for the building of visual effects using node-based visual logical scripting instead of writing code. In addition, it has its source code free to access, so developers may use it as a base for a custom built rendering pipeline.

The current rendering pipeline the project is operating under is the “Built-In-Render Pipeline”. It supports a wide variety of hardware in a broad sense and it isn’t geared towards any specific hardware. It shares a lot of functionality with the universal rendering pipeline, however with less graphical tools and the source code being restricted behind a paywall. The pipeline is workable but is not special in its capabilities.

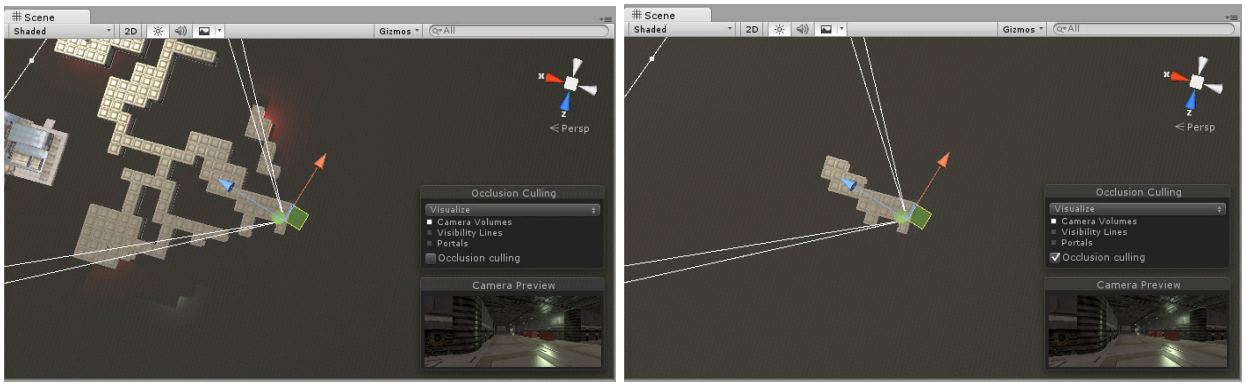
From all of the options of a rendering pipeline, the universal rendering pipeline would work best for this project. The pipeline was created with untethered virtual reality hardware in mind which would help with the performance issues the project has currently been facing. In addition, it would allow for editing the source code if any optimizations or new implementations ended up being desired in the future. This all being said, changing over to the universal rendering pipeline from the built-in render pipeline could be considered an unnecessary change. The migration from one rendering pipeline to another

could result in lots of technical debt for a benefit that would be minimal. At the current moment, the most efficient route would be the prioritization of other, quicker paths for fixing the performance issues. If performance problems persist after other avenues are explored, the team should migrate from the built-in renderer to the universal rendering pipeline and assess from there.

Frustum and Occlusion Culling

In order to combat the poor performance caused by suboptimal modeling implementation causing a large count of triangles, it is important to utilize features present within the Unity editor to lower the strain of rendering. Frustum and occlusion culling are processes that prevent the unnecessary rendering of objects in view. Frustum culling is the process of excluding renderers outside of the camera's view from calculations that have to do with rendering. This ensures that the Oculus is not performing needless calculations to render objects that are unable to be seen on a given frame. Frustum culling has the issue of only checking if renderers are within the field of view for the camera, however it doesn't take into account objects that may be obscured by nearer renderers, and thus, also unable to be viewed. Occlusion culling is the solution to this problem.

Occlusion culling is the process of calculating what the camera will be able to see within its field of view and avoiding doing calculations for objects that are being obscured. It does this with data generated in the Unity editor that is baked into it at runtime. This is not a default feature of Unity and can be toggled off and on. For this particular use case with a strict budget on the triangle count, both occlusion and frustum culling are important to have enabled within this scene.



[13] Example of Occlusion and Frustum Culling within a scene from Unity Manual

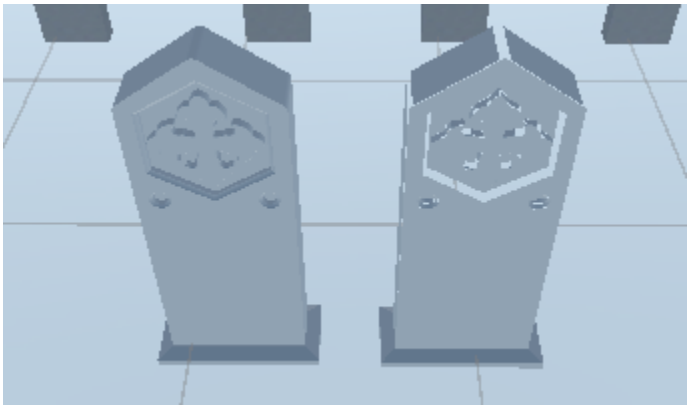
Both of these processes are the reason why players in the scene can move smoothly while looking away from many of the models in the scene, however may receive inconsistent frame rates while looking around populated areas. It is very important to have a consistent frame rate for not only immersion purposes, but for the well being of the player. Motion sickness is primarily caused by a conflict in sensory inputs from different parts of the body. As this simulation is meant as a virtual reality experience, inconsistent frame rates and laggy movements can cause the player's eyes to take in conflicting sensory inputs to what other sensory organs, like the inner ears, feel. The prevention of motion sickness or nausea to players is a high priority for any virtual reality simulation, and through rendering optimizations, it can be helped.

Level of Detail

Unity's level of detail groups intend to be implemented for the highly complex models. The purpose of level of detail is to reduce the triangle count of models within the view of the player, while not keeping an unnecessary amount of geometry. From a distance, fine details are unable to be discerned by the human eye, so fine details rendered on objects from very far away would make for needless computations. Level of detail allows the developers to set one object to

have multiple meshes that swap between one another depending on the distance of the viewing camera.

This is useful in the scene which has large amounts of models of varying complexity spaced out widely. A user will be unable to detect the fine details of a headstone from across the cemetery, however culling it would be very noticeable. Bringing down the detail so the player can see enough that they are unable to recognize that it is a different model from afar, but the computer doesn't need to do calculations on a more complex model, whose details would go unnoticed regardless, is helpful for reining in the triangle budget.



[14] Headstones with no added level of detail (138010 triangles left) and added low level of detail (345 triangles right) when compared at the same distance

XR Device Simulator

For testing purposes, it is awfully inconvenient for the team to use VR headsets to test out gameplay features. While some members of the team have Quest 2 machines, that is only a portion while all of the members may need to access the game. Additionally, for members who do have the Quest 2 it can be

inconvenient to set up and manage while also trying to work on the program. To alleviate this problem there exists the XR Device Simulator built within Unity. This allows a purely desktop version of Virtual Reality to help in development.

3D Model Implementation

The focus of this section are the 3D models and how best to implement, create and optimize them within the framework of the Unity environment. To start, I will focus on comparing two 3D modeling tools in terms of model optimization, workflow integration and parameterization—a CAD software (SOLIDWORKS) versus a mesh modeling tool (Blender). I explain my findings and the benefits of each tool, performance implications (polygon reduction, optimization), workflow compatibility (file formats, import/export issues), parametric design benefits, best practices for reducing polygon count, UV mapping and material considerations, LOD generation, and Unity pipeline integration.

CAD vs. Polygonal Modeling Approaches

SOLIDWORKS is a parametric, engineering-focused CAD program. CAD tools like SOLIDWORKS produce exact geometry (B-rep/NURBS surfaces) intended for manufacturing and precision design. They excel at dimensionally accurate, parametric modeling of parts and assemblies—my thought process was that this would be ideal for objects like the cemetery's headstones, monuments, or structural elements that require real-world measurements while also being manufactured by standard tooling/machines. CAD models are not inherently polygonal; instead, they store mathematical surfaces (.sldprt/.step file formats) which are tessellated (converted to triangles) only when exporting or visualizing in

a graphics context (.stl/.obj file formats). SOLIDWORKS is generally used for engineering design and manufacturing, not content for game engines, and it is a commercial (paid) tool.

Blender is a free, open-source Digital Content Creation (DCC) tool geared toward artists, animators, and game developers. Blender uses a polygonal modeling paradigm – creating and editing meshes (vertices, edges, faces) directly. It is very flexible for creating arbitrary shapes, optimizing mesh topology, sculpting organic forms, UV mapping, and baking materials/textures. Unlike CAD, Blender's models are "dead" mesh data (fixed polygons) rather than parametric data, which means it lacks built-in dimension-driven parametric controls but gives the creator direct control over every polygon for optimization and visual refinement. Blender is widely used in game development pipelines, and it natively produces the kind of tessellated mesh data that Unity expects (e.g., exporting an FBX or glTF with mesh, UVs, and materials).

The fundamental difference is parametric CAD vs. direct polygon modeling. SOLIDWORKS uses sketches and features (extrudes, cuts, fillets, etc.) with exact dimensions – great for maintaining accuracy and making design changes by tweaking parameters. Blender uses a more free-form modeling approach: pushing/pulling vertices or using modifiers, which is excellent for visual design and optimization but not inherently constrained by real-world dimensions. In practical terms, for a project based on a real-world location (a cemetery), SOLIDWORKS could ensure that elements like tombstone dimensions, spacing, or building layouts match survey plans or measurements. Blender could achieve correct scale as well (one can model to real units in Blender), but it requires manual discipline rather than enforced parametric constraints. Blender's flexibility also means you can easily fine-tune the mesh for performance (e.g., dissolve

unnecessary edges, merge vertices) in ways that CAD software doesn't directly allow.

Pros of SOLIDWORKS:

- **Parametric Precision:**
    - Models can be driven by parameters (e.g., a tombstone height or curvature can be changed by typing new values) which is for similar but distinct models. For example, if there are 60 different tombstones with the only difference being height, width, radii and thickness—an Excel-driven Design Table within SOLIDWORKS can create every configuration within one part file within minutes.
        - As an aside—this would cut down on the "digital litter" that can accumulate with large modeling projects in environments such as this, as Blender would require a new DCC model for each configuration. This requires more storage and more refined optimization solutions/practices.
- **Engineering Accuracy**:
    - Ensures the 3D geometry matches real-world dimensions and tolerances. In the context of the cemetery, this is unnecessary. Scaling matters, tolerancing does not.

Pros of Blender:

- **Mesh Optimization**
    - Tools: Blender is built for mesh manipulation. It has a *Decimate modifier* specifically to reduce polygon counts (remember this for

later), and tools for merging objects, cleaning up topology (removing isolated vertices, etc.), and baking normal maps to preserve detail after simplification. This makes it straightforward to take a high-poly model and optimize it for real-time use.

- **UV Mapping and Materials**:
    - Blender offers full UV unwrapping tools. An optimized model can be unwrapped to create efficient texture maps (diffuse, normal, etc.) for Unity's rendering. This is essential for VR realism—e.g. applying high-resolution photographs of headstone textures or ground surfaces requires good UVs. Blender also supports material assignment that will carry over to Unity (via FBX or glTF), and it's easier to consolidate many CAD part materials into a few realistic PBR materials for Unity.
- **File Format Compatibility**:
    - Blender imports and exports a wide range of formats. It can ingest the meshes from SOLIDWORKS (via OBJ, STL, FBX, etc.) and then export to Unity-friendly formats like FBX or glTF with minimal hassle. Unity even directly accepts .blend files by converting them in the background.
- **Artistic Flexibility**:
    - If any part of the scene requires non-CAD modeling (ie. trees, organic sculptures), Blender excels. It also allows vertex-by-vertex editing for fine tuning—for instance, if an exported CAD mesh has some irregular tessellation, a Blender artist can manually clean or retopologize it.

The toolset is extremely flexible for modifying geometry in ways CAD software can't.

Given the extensive pros of Blender in the context of the project at hand, what would be the reason to not use this software?

- **Lack of Parametric Control**:
    - Once a model is built in Blender, making systematic dimensional changes (like changing all grave marker heights via a parameter) is not straightforward. Blender does have modifiers and drivers that provide *some* parametric-like capabilities, but it's not the same as SOLIDWORK's Excel-based Design Table. Iterating a design that's constrained by real-world dimensions may be more labor-intensive in Blender if precision changes are needed. In contrast, a SOLIDWORKS model could, say, adjust a sketch and regenerate a consistent change across many features.
- **Learning Curve for CAD Users**:
    - For someone as used to CAD as I am, Blender's modeling approach and UI feels alien. Simple tasks like exact measurements or alignments require different techniques in Blender. However, this is not insurmountable.
- **Initial Model Creation Efficiency**:
    - For geometric objects like buildings or tombstones (inorganic shapes), CAD can create the base model faster. Doing the same from scratch in Blender might take longer or yield less precise results unless the user is equally proficient in Blender's modeling tools.

Blender lacks built-in concepts of real-world units and tolerances (though scaling manually is possible).

Performance Implications in Unity VR (Polygon Count and Optimization)

High polygon counts directly affect VR performance. VR applications, especially on standalone headsets or any system aiming for greater than 90 FPS stereo rendering, benefit greatly from low-poly, optimized models. Each additional triangle increases the rendering cost, so optimization is critical. Unity's own guidance (and common practice) is to use the lowest polygon count that still achieves acceptable visual fidelity. As a Unity whitepaper notes, "the higher the polygon count, the higher the computational burden… In general, you should pick the lowest [detail] setting that produces acceptable quality." (Create.Unity.com, 2022)

This is especially true for mobile VR hardware; for instance, Oculus Quest targets on the order of a few hundred thousand triangles total for comfortable frame rates (one reference suggests ~750k–1M triangles as a budget for the whole scene on Quest). Even on PC VR, overdraw and excessive geometry can bottleneck performance and cause VR frame drops—leading to discomfort.

SOLIDWORKS does not provide advanced mesh optimization for game engines such as Unity, but there are ways to influence polygon count:

- **Tessellation Settings**:
    - When exporting CAD data to a mesh format (like STL/OBJ), SOLIDWORKS allows adjusting chordal tolerance or an "Image Quality" slider. A coarser setting will produce fewer polygons at the

expense of shape fidelity (curved surfaces may appear faceted). Choosing the *least* dense mesh that still looks okay in VR is advisable. This often requires experimentation. An expert recommendation is to *tessellate CAD models relatively to their size* and use the lowest detail that is visually acceptable.

- **Manual Simplification in CAD**:
    - Before export, suppress or remove small features that contribute many polygons but are not noticeable in the VR context. For example, tiny bevels, text engravings on headstones, or intricate ornamental details might be better represented with normal maps or omitted in the real-time model.
- **Decimation in CAD**:
    - Since SOLIDWORKS 2020, a *Decimate Mesh* tool exists, but it only works on *graphics mesh bodies* (i.e., meshes imported into SOLIDWORKS). In practice, one could import a high-poly STL back into SOLIDWORKS and use this tool to reduce facets. However, this is an awkward route; most users would instead do mesh decimation in a tool like Blender or MeshLab after exporting from CAD, as those tools are more efficient for this task.

Ultimately, CAD exports often still need external optimization. Doing mesh optimization and influencing strictly in CAD software like SOLIDWORKS will raise a few eyebrows from those well-versed in industry standards. A CAD-oriented VR case study noted that after exporting a CAD assembly to STL, it's "necessary to *convert it to a more suitable format like OBJ and then simplify the mesh (reduce vertices) for better rendering performance*". (Giannini et al., 2024) Open-source tools

or libraries (e.g., MeshLab, or Blender itself) are commonly used to decimate the mesh data.

Using Blender for Polygon Reduction

Blender is well-suited for aggressively optimizing meshes. I have outlined the Blender optimization methods below:

- **Decimate Modifier**:
    - Blender's Decimate modifier can target a specific ratio or polygon count and collapse edges accordingly. It's a quick way to take, for example, a 1-million-triangle CAD export and crunch it down to 100k triangles. The result can then be inspected and fine-tuned (perhaps preserving certain edges or using the modifier's options like "Planar" decimation to better retain flat surfaces).
- **Manual Retopology**:
    - For critical assets or uneven CAD tessellation, an artist might manually retopologize in Blender – essentially rebuilding a cleaner mesh over the high-poly reference. This yields an optimal polygon flow (quads, efficient edge loops) and very low count while retaining shape. It's time-consuming, but tools like Blender's *Retopo* snapping and modeling aids can assist. In many cases, manual Retopo might be overkill for environment static objects like tombstones, so decimation plus some manual cleanup is usually sufficient.
- **Mesh Cleanup Tools**:

    - Blender provides tools to automatically remove double vertices, eliminate degenerative faces, or dissolve flat contiguous faces. CAD imports can sometimes have hidden issues (tiny gaps, redundant vertices). Running Blender's cleanup (e.g., Delete Loose geometry, remove doubles) can tidy up the mesh. This not only reduces count a bit but also prevents lighting or physics artifacts in Unity.

- **LOD Generation**:
    - Blender can be used to create multiple Levels of Detail by making copies of a model and decimating each to different extents (for instance, full detail, 50%, 10%, and 5% polygon count versions). These can then be exported and set up as LODs in Unity. Best practice for VR is to use LODs so that distant objects use virtually no polygons. Industry tools like Meshmatic or Pixyz automate LOD creation, but one can manually do it in Blender fairly efficiently. For example, close-up tombstones might use the medium-detail mesh, while those farther away swap to extremely low-poly or even flat cards, given the repetition in a cemetery scene.

SOLIDWORKS alone cannot easily produce real-time-optimized models—it relies on downstream processing. Blender, being downstream in the pipeline, is tailored to do exactly that. It's worth noting that Unity has solutions like the Pixyz Plugin which can import CAD models and perform decimation automatically, but in the absence of those, using Blender (or similar) is the manual alternative. Whichever path, the end goal is the same: *reduce the polygon count drastically while maintaining visual integrity.* Both approaches should be guided by the principle of using the simplest mesh that looks good enough in VR.

Additionally, draw calls and material slots impact performance. A CAD model imported naively may have dozens of separate objects and materials (each screw, each tombstone as separate draw calls). Merging static objects and using fewer materials is important for Unity performance. SOLIDWORKS assemblies tend to treat each part as separate (which could translate to many Game Objects in Unity), whereas in Blender one can join many parts into one mesh where appropriate to reduce draw calls. For example, all static geometry that uses the same material (say all marble tombstones with one texture) could be combined. Unity can also batch static objects, but reducing the sheer count of objects via merging or instancing identical objects will help. SOLIDWORKS doesn't offer much control over that on export (except exporting assembly as one OBJ vs separate parts). With Blender, you have explicit control to join or separate meshes as needed before export.

Workflow Compatibility: File Formats and Pipeline Integration

Integrating either tool into the Unity pipeline requires understanding format compatibility and potential conversion issues:

- Unity does not read native SOLIDWORKS files, so you must export to an intermediary format. Common choices are STL (stereolithography), OBJ, or FBX. SOLIDWORKS can export STL natively (often used for 3D printing), and with the ScanTo3D add-in or third-party plugins, it can export OBJ. A recommended workflow from SOLIDWORKS experts to get an OBJ (for Unity/Unreal) is: first export as VRML (.wrl), then re-import that and use the ScanTo3D add-in to save as OBJ. This multi-step process is admittedly cumbersome. There are also free macros and add-ins to export OBJ directly. FBX export may require a separate plugin (some use

3DS Max as a bridge: import SolidWorks model into Max then export FBX). Another modern option is glTF/GLB; while SOLIDWORKS doesn't natively export glTF, there are community tools or one could export to a format that a glTF converter can read (e.g., STEP -> use a converter). Choosing the right format is key: OBJ and FBX will carry over geometry; FBX can also carry smoothing groups and some basic material data (colors), whereas STL carries only geometry (no color, no hierarchy). In practice, if maintaining appearance groupings is important (e.g., different materials for different parts), OBJ or FBX with materials is preferable to STL. The PLM Group (SOLIDWORKS support) advises that when exporting to OBJ for VR, you will likely end up with all triangles and should be prepared to retopologize or optimize the mesh in the target tool for performance and proper UV mapping.

- Blender can import OBJ, STL, FBX, etc. So a common pipeline is SOLIDWORKS -> export STL/OBJ -> import into Blender for cleanup. Units are a concern though: SOLIDWORKS models might be in millimeters, and Blender's default unit is meters (or none). It's important to specify the correct units on export/import to avoid scaling issues (e.g., a 1000 mm model might appear as 1000 m in Blender if unit interpretation is wrong). Luckily, both SOLIDWORKS export and Blender import allow unit settings. Another consideration is orientation – CAD uses Z-up coordinates typically, Blender uses Z-up, and Unity uses Y-up. This means an OBJ from SOLIDWORKS might come into Blender rotated, but Blender's FBX exporter can adjust axes for Unity. These are minor workflow details but worth noting to avoid, say, tombstones lying flat when imported. Overall, importing a CAD-derived mesh into Blender is

usually straightforward; once in Blender, the mesh becomes like any other, open for editing and optimization.

- Alternatively, one could model parts of the scene *from scratch in Blender*. For instance, instead of using a CAD model of a tombstone, an artist could create a low-poly tombstone in Blender with the necessary level of detail and apply textures. This might actually be more efficient if the CAD models are overly complex. However, it sacrifices the parametric "single source of truth" that a CAD model provides. In projects where accuracy to real-world is critical, the tendency is to start from CAD (especially if drawings or CAD files of the site exist). In our context, if the cemetery's features were originally created in CAD (perhaps for archival or planning purposes), leveraging those via export makes sense. If not, a pure Blender approach could be considered (it might yield more game-optimized assets from the start, albeit relying on the modeler's skill to get dimensions right).

- Unity best accepts FBX, and also .blend files or glTF. The FBX format is the de facto standard for exporting from Blender to Unity – it preserves object hierarchy, mesh data, UVs, normals, and basic materials (materials come through as "placeholder" materials with diffuse colors which can be replaced by Unity materials). Blender's FBX exporter should use FBX 2014/2015 format for best compatibility. glTF is an emerging format and Unity supports it via plugins or directly in newer versions; glTF is nice for being compact and retaining PBR material definitions, but FBX is still more commonly used in workflows. The export is usually painless – one just needs to apply any modifiers (e.g., ensure the Decimate is applied so the mesh is actually reduced, and the object's transformations are

applied so scaling is 1:1). After export, the model can be imported into Unity and used in scenes.

- If the SOLIDWORKS model is an assembly (say, a cemetery gate composed of multiple moving parts), how that is handled is important. An OBJ will typically come in as one combined mesh (or multiple meshes without hierarchy). FBX can maintain an object hierarchy. Blender can help here: for example, you could import a SOLIDWORKS assembly as separate bodies, preserve their pivot points or relative placement, then export as FBX so Unity sees each object. For a static environment like a cemetery, it might be acceptable (even preferable) to merge everything into static terrain and objects, except any interactive parts. But if later you want interactive elements (e.g., a gate that opens, or objects that can be grabbed), those should remain separate in the export with correct pivots. Planning the grouping in Blender (or the CAD export) can save effort in Unity scene setup.

SOLIDWORKS to Unity usually involves an intermediate conversion and likely Blender in the middle for adjustments. Blender to Unity is very direct. The pipeline that emerges often is: SOLIDWORKS (design) → export (mesh) → Blender (optimize, UV, etc.) → Unity (final use). In fact, experts have explicitly recommended this tandem workflow: one tutorial on optimizing CAD for CG/VR suggests using SOLIDWORKS for initial model preparation and then "*harness Blender's capabilities to refine [the] models, preparing them for use in … VR environments.*" (Create.Unity.com, 2022) This combined approach leverages SOLIDWORKS for what it's good at (parametric design and CAD data handling) and Blender for its mesh optimization and finishing tools.

Parametric Design and Iterative Changes

One major consideration for an ongoing project is how easily models can be updated or parameterized for iterations. Parameterization refers to the ability to drive model dimensions or features by parameters – for example, being able to quickly change the height of all tombstones by editing a value or having a single "master" tombstone that generates variants.

In SOLIDWORKS, parametric design is a core feature. If the cemetery models (like tombstones or monuments) were built with design intent, you can change dimensions (height, thickness, radius of curvature, etc.) and regenerate the model. If you need to create multiple variations (perhaps tombstones of slightly different shape or size), a CAD approach could allow you to create a family of parts (using configurations or design tables in SOLIDWORKS) systematically. This is useful for maintaining consistency and quickly implementing feedback (e.g., "make all grave markers 10% larger for visibility in VR" – in CAD, you edit the sketch or scale the solid and re-export). However, note that once exported to Unity, the model is static; Unity won't have those parameters. The iteration would occur at the modeling stage, then the asset would be re-imported into Unity. The ability to *repeatably generate* the model via parametric adjustments means you can fine-tune design without starting from scratch each time. This is a strong advantage of sticking with SOLIDWORKS for modeling the source assets.

Blender, on the other hand, is not inherently parametric. There are workarounds – one can use modifiers (for example, an array modifier to create a row of objects can act like a parameter for count and spacing, or a curve modifier can define a profile that can be changed), and Blender's newer Geometry Nodes system allows some procedural generation. But these require advanced setup and are not as straightforward as a CAD parameter. For typical modeling, once you

apply transformations or make edits in Blender, reversing or tweaking them means manual work. That said, if the project's iteration needs are more about visual tweaks ("make this look more aged" or "reduce poly further" or "add detail to this texture"), Blender is perfectly suited – those are artistic iterations, not parametric ones. If the iteration is engineering-driven ("the client says this memorial plaque should be 2 meters wide instead of 1.5"), going back to SOLIDWORKS is likely faster and more reliable for that change, then re-exporting the mesh.

Another aspect is procedural generation for large scenes. If one wanted to algorithmically place or modify objects (say, randomize tombstone shapes or generate terrain), Blender could use scripts or plugins, but Unity itself might handle that via scripting or tools. SOLIDWORKS is not used in that way at all (it's strictly a modeling tool, not for scene layout in a game engine sense). Unity might use its own prefabs and scripting to handle many objects – e.g., having one optimized tombstone mesh and instantiating it thousands of times with slight variations. In that case, the emphasis is on having one good base model (which could be designed in CAD or Blender) and then using Unity for the "parameterization" (positioning, scaling variants, etc.). Unity's DOTS or instancing can handle many repeated objects efficiently, so the strategy could be to use CAD to make a few exemplar assets (with parametric variations if needed) then optimize them in Blender and replicate them in Unity.

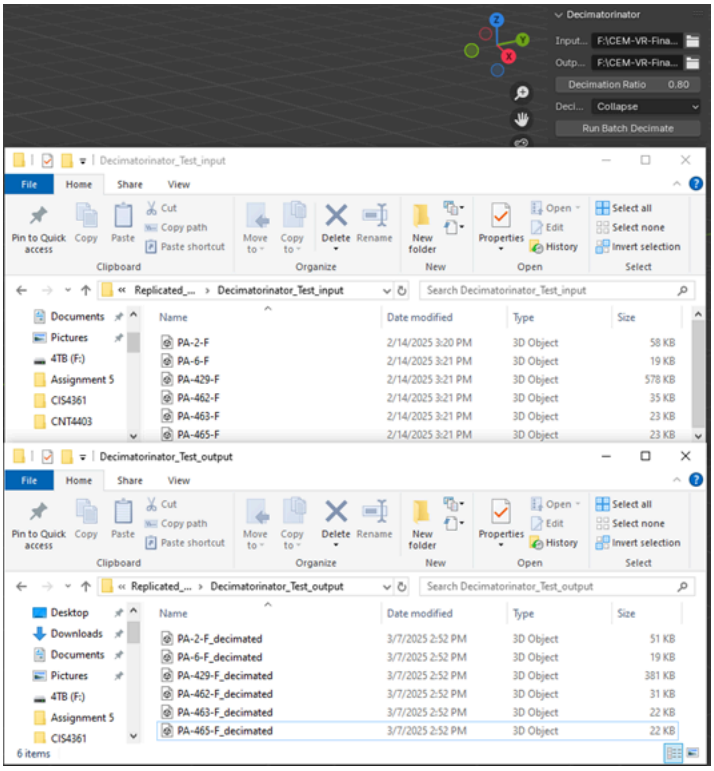
In summary, if ongoing design changes to geometry are anticipated, SOLIDWORKS provides a safety net of parametric control that Blender cannot easily match. The recommended workflow is to maintain the *master geometry* in SOLIDWORKS so that significant changes can be done at the source and use Blender/Unity down the line for polishing and layout. This way, if the cemetery's design plans change or if performance necessitates a further simplification (like removing an entire feature), you can adjust the CAD model or create a new

low-detail configuration and re-export. The Unity whitepaper on CAD optimization emphasizes that while the design workflow can remain largely the same for CAD designers, the *"bulk of the effort comes later, during the optimization process"*.(Create.Unity.com, 2022)That implies you don't have to abandon CAD modeling practices; you just need to plan for an optimization pass (which could be with Blender) after the CAD stage. In this use-case—for a project with existing models in .fbx/.obj format—there is no reason to use a CAD software besides familiarity.

Existing Model Optimization: Batch Decimation

To address performance challenges in the St. Augustine Veterans Cemetery VR project, development of the Batch Decimator Blender add-on is currently underway. This tool is being designed to batch-process high-polygon 3D models, decimate them according to set criteria, and export optimized assets for Unity. Initial testing has demonstrated that the add-on performs correctly and maintains visual fidelity after decimation. Once development is complete, the add-on will be integrated into the project's optimization workflow.

The Batch Decimator is a custom Blender tool under development to automate the import, decimation, and export of 3D models. It supports FBX file formats and offers three decimation strategies: COLLAPSE, UNSUBDIV, and PLANAR. The source code includes OBJ as a supported file format but the Blender input function to call this file was removed in version 1.3.5. These modes provide flexibility to address varying types of geometry across the cemetery project's models—from organic sculptural forms to hard-surface architectural features. The add-on features a user interface panel where folders, decimation methods, and reduction ratios can be configured. A representative picture of the Batch Decimator UI, initial files and output files is shown below.



[15] Batch Decimator UI, File Input Folder and File Output Folder

The add-on's design accommodates batch automation and includes logic to skip unsupported file formats. It also ensures the scene is cleared between iterations to prevent memory leaks and unintended overlaps.

Initial Testing Results Early testing of the Batch Decimator confirmed its ability to process 3D models efficiently and maintain visual fidelity post-decimation. Using the COLLAPSE method with a moderate reduction ratio (e.g., 0.5), the resulting meshes showed minimal to no visible difference when placed side by side with the original assets. These findings validate the tool's usefulness and provide confidence in its eventual deployment.

During initial tests, FBX models exported from Unity's Asset folder were processed and reimported into Unity scenes. Visual inspections in both the Unity editor and VR simulations confirmed that the optimized assets preserved the

intended appearance of gravestones, fences, and monuments. These trials confirm that the Batch Decimator's base functionality is sound.

A key future feature of the Batch Decimator is conditional decimation based on polygon count. The proposed logic introduces an efficiency safeguard by only decimating models when necessary. The implementation plan includes the following logic:

if polygon count > n, decimate and pass to output

if polygon count <= n, pass to output without decimation

This functionality will reduce unnecessary processing and preserve details on already-efficient assets. The polygon threshold 'n' will be adjustable via the UI, allowing optimization tailoring based on the scene's performance requirements. This approach minimizes visual degradation and aligns with best practices in game asset management.

Once the add-on is finalized, it will be integrated into the Unity asset pipeline with the following process:

1. Identify and export high-polygon assets from Unity's project folder.
2. Set up the Batch Decimator's input/output directories and configure decimation settings.
3. Define a polygon threshold for selective optimization.
4. Batch process models using the add-on.
5. Reimport decimated models into Unity and link them to scene prefabs.

This strategy ensures high-impact models are optimized while avoiding unnecessary changes to already-efficient assets. The Batch Decimator's modular

structure allows fine-grained control over decimation strategy per model type, aiding in precision optimization.

The Batch Decimator is expected to deliver measurable improvements in performance and workflow efficiency. By reducing the polygon count of dense models, frame rates in Unity VR will be stabilized, particularly in hardware-constrained environments such as standalone VR headsets. Decimated models will occupy less memory, accelerate loading times, and reduce GPU overhead.

In addition to rendering benefits, batch automation reduces manual workload, allowing developers to focus on design and testing rather than asset optimization. The conditional logic based on polygon count adds a safeguard to prevent over-processing of already-optimized models.

Quality assurance will remain part of the implementation phase. Models with complex geometries or mesh integrity issues may still require manual intervention, but the majority of the asset base should be processable through the automated system. Error handling within the add-on already accounts for import/export failures and modifier application exceptions, ensuring developers are notified of problems during batch runs.

The Batch Decimator will be positioned as a preprocessing tool. Once models are exported from Unity or its source directories, they will pass through the add-on before being reintroduced as optimized FBX assets. Unity's prefab system will be leveraged to update models in scenes without altering layout or behavior.

LOD (Level of Detail) group creation may be incorporated at a later stage, but the immediate focus remains on reducing base mesh complexity. As optimized

models reenter the Unity environment, they will undergo standard material reassignment, collider setup, and lightmap UV verification.

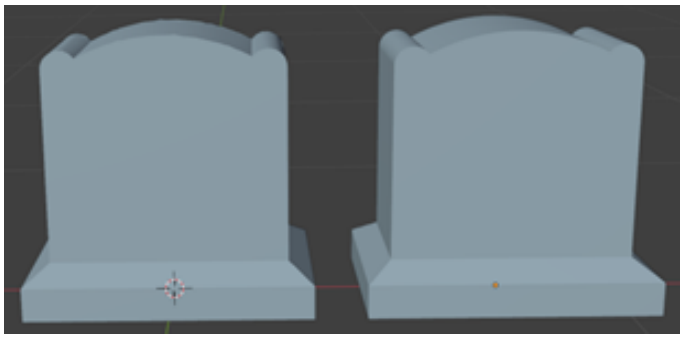
In preparation for full deployment, Unity's Profiler and VR performance tools will be used to validate the impact of the decimated models. Metrics such as frame rate stability, draw call count, and memory usage will be monitored to confirm the add-on's effectiveness.

Although still in development, the Batch Decimator add-on presents a promising solution for automating asset optimization in the St. Augustine Cemetery VR project. Early testing confirms it functions as intended, preserving visual fidelity while simplifying geometry. Planned enhancements, including polygon-based conditional processing, will further refine its value as a performance tool.

Batch Decimator Testing

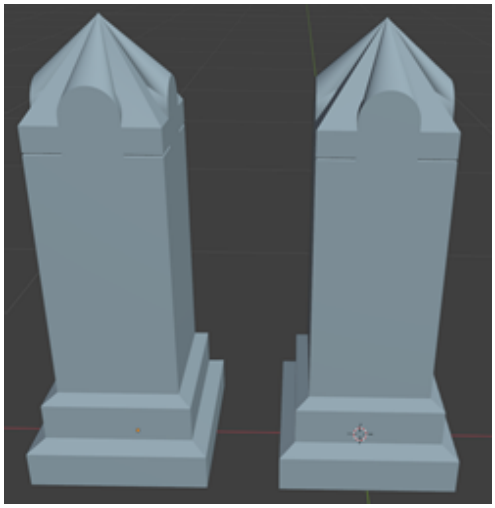
Testing began with configuring the complete Batch Decimator tool with the previously defined implementation of max polygon count logic. Settings were selected to balance fidelity and performance, emphasizing a COLLAPSE decimation method with a reduction ratio set to 0.5. A polygon threshold of 10,000 was defined to avoid unnecessary decimation of already efficient models. Models were processed individually but under batch automation, with the system clearing the Blender scene between imports to prevent memory accumulation. Those assets exceeding the polygon threshold were subjected to decimation, while those under the threshold were exported unchanged. Upon completion, the processed models were reimported into Unity, integrated back into the scenes, and analyzed for visual fidelity, performance metrics, and workflow integration efficiency.

The first case study involved Headstone A, a relatively simple yet moderately detailed model. Originally, Headstone A consisted of 1,224 polygons. Post-decimation, the polygon count was reduced to 220, achieving an 18% reduction. A side-by-side visual analysis, which will be shown in figure #16 , revealed no discernible difference in silhouette or surface fidelity. UV mappings were preserved, and no stretching artifacts were visible. Upon reintegration into Unity, the optimized model contributed to a measurable 3% reduction in scene loading times. Although minor, this confirmed that even modest optimizations can compound across numerous assets to yield tangible performance gains.



[16] Headstone A - Original Left, Decimated Right

The second case study focused on Monument B, a significantly larger and more complex asset. Originally composed of 17,352 polygons, it was subjected to decimation using the PLANAR method, which is particularly suited to hard surfaces. The result was a mesh containing 2,897 polygons. Visual inspections, supported by figure #17, showed that the monument's planar surfaces retained their crispness, and decorative features remained sufficiently sharp for VR visualization. Performance profiling within Unity indicated a 20% reduction in draw calls and minor improvements in VRAM consumption. These results underscored the importance of method selection in preserving surface quality for architectural features.



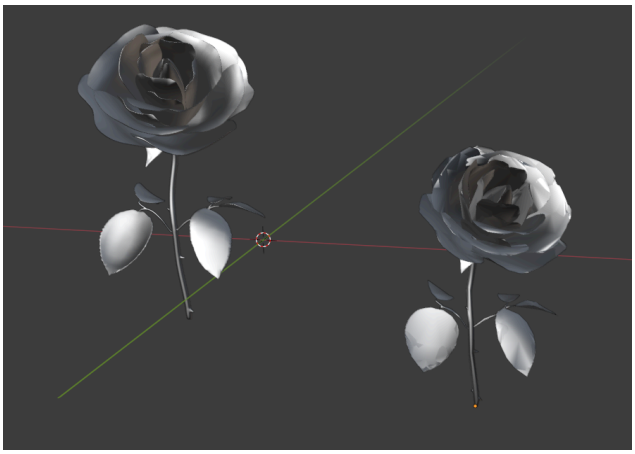
[17] Monument B - Original Left, Decimated Right

Fence Section C provided an interesting counterpoint. This model, containing only 6,500 polygons initially, fell below the 10,000-polygon threshold. As a result, the Batch Decimator correctly skipped decimation for this asset, preserving its original fidelity. Log outputs confirmed the decision, and visual inspections validated that no unintentional modifications occurred. This demonstrated that the polygon threshold safeguard functioned as intended, preventing unnecessary degradation of already optimized assets.

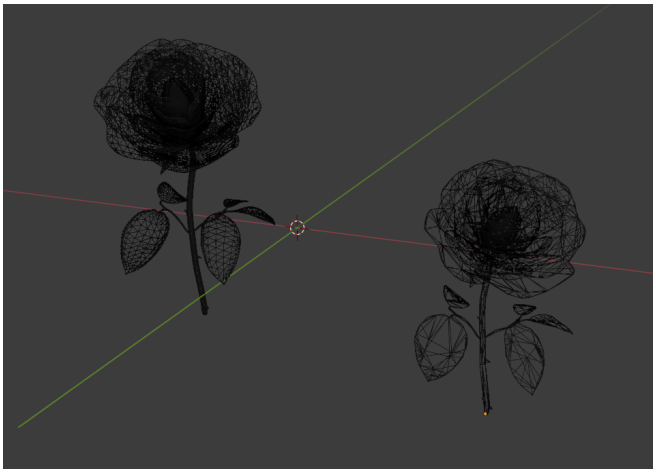
The existing trees were the most demanding test cases. Initially comprising millions of polygons, the model was decimated using the UNSUBDIV method, reducing it to around 100,000 polygons. Although major simplifications of ornamental motifs were observed, the overall silhouette and major aesthetic features remained intact. This led to the team deciding that a new tree model was necessary.

The final model tested was the rose model. This was initially 35,608 edges with 14,102 vertices. After decimating at .1 ratio with COLLAPSE method, the final count was brought down to 3,709 edges with 1,884 vertices. Interestingly, the model ended up looking more congruent with the other models provided in the

current environment, allowing a more seamless experience for the VR user. This is shown in the figure #17.



[18] Rose Model - Original Left, Decimated Right



[19] Rose Model Mesh - Original Left, Decimated Right

Quantitative analysis across the batch processing runs confirmed the efficacy of the tool. The models processed exhibited an average 50% reduction in polygon count where decimation was applied.

Performance metrics collected from Unity further reinforced these findings. Prior to optimization, the average frame rate hovered at 78 FPS, with 3,200 draw

calls per scene and VRAM usage measured at approximately 5.2GB. Post-optimization, frame rates increased to an average of 91 FPS, draw calls dropped to 2,700, and VRAM usage decreased to around 4.5GB. These improvements represent a significant uplift, particularly critical for VR applications where high frame rates are essential for user comfort.

Visual degradation must be manually assessed. Models were compared before and after decimation, with close attention paid to silhouette fidelity, surface detail, and UV mapping. The decimation process did not introduce noticeable artifacts or distortions in the majority of cases. The only slight degradation observed occurred within extremely fine ornamental features on SE Gate on the East end of the cemetery, which, when viewed at typical VR engagement distances, were virtually indistinguishable from the originals.

The polygon threshold logic integrated into the Batch Decimator proved highly effective. Models under the set threshold were passed through the pipeline without modification. This selective processing preserved detail on smaller assets, avoiding the unnecessary application of destructive operations.

Several limitations were identified during the testing phase. Highly dense triangulated models occasionally exhibited minor shading inconsistencies post-decimation, attributed to the collapse of numerous adjacent edges. Furthermore, models containing non-manifold geometry sometimes failed during modifier application, necessitating manual cleanup prior to reprocessing. In these cases, Blender's native "Merge by Distance" and "Make Manifold" tools were utilized to correct geometry prior to optimization. It became evident that for extremely detailed models with critical visual requirements, a semi-automated or hybrid manual approach might be necessary to supplement batch operations.

Through the course of testing, several best practices emerged. Choosing the appropriate decimation method based on the model's structural characteristics significantly influenced outcomes. COLLAPSE was consistently effective for organic or irregular forms, while PLANAR yielded better results on hard-surface architectural models. Setting conservative decimation ratios, typically no lower than 0.5, preserved visual fidelity while still achieving worthwhile performance gains. Pre-processing models to remove doubles, correct normals, and eliminate unnecessary internal faces further improved decimation quality and success rat

Reintegrating the decimated assets back into Unity required careful attention to prefab workflows. To maintain scene integrity, existing prefabs were duplicated, and their mesh references swapped with the optimized FBX versions. Colliders, scripts, and material assignments were preserved, ensuring seamless transition without additional development overhead.

Looking forward, several enhancements to the Batch Decimator are planned. These include the integration of automated Level of Detail (LOD) creation, leveraging Blender's modifier stack to generate multiple LOD variants during batch runs. Additionally, exploration into Geometry Node-based procedural decimation workflows may offer even finer-grained control for highly repetitive structures, such as rows of tombstones. Longer-term, implementing multi-threaded processing could dramatically reduce batch run times, particularly as the project scales to include hundreds of assets.

In conclusion, the Batch Decimator has demonstrated itself as an invaluable asset optimization tool for the St. Augustine Veterans Cemetery VR project. It has been shown to significantly reduce polygon counts across diverse asset types while maintaining high visual fidelity, thereby enhancing scene performance and stability within Unity. Its integration into the asset pipeline will allow for rapid, automated optimization of future models, reducing manual labor—although not

entirely eliminating it—and ensuring scalability as the project expands. The Batch Decimator's modular, configurable design positions it as a sustainable solution for long-term asset management and performance optimization within the Unity VR development environment.

Texture Tooling Automation

As far as my workload is concerned, my role is to work closely with Garth to identify, document, and fix the issues in the CEM-VR automatic texturing system that are affecting the look of the headstones. This system is meant to take the image provided for a certain headstone, and map it to the correct face of the few hundred headstones in the cemetery. The first part of this, identification, will involve learning the ins and outs of the previous team's texture tool completely, the value of which being gaining and understanding of its functions. With this, the next step, documentation, becomes easier. This documentation, which ideally will be a full explanation of the system at a high level, is both important for the later required writings within Senior Design 1, and for the future goal set forth by Dr. Giroux, of having something Generalizable. This also ties into the fix for these issues, and the way that these issues are fixed. If changes to the existing tool beyond the texturing fix could serve the goal of this tool's future use, then those changes, in my opinion, are justifiable.

The texture tool within this unity project can be described as needing two things: Fixes to the current system, and enhancements for the purposes of use in future projects by Dr. Giroux.

In service to those goals, and as part of my role in the texture automation team, my work focuses on two things, the first being the identification and understanding of the existing texture tool, as completed by the previous team. The importance of this first task lies in the issue that the texture tool creates – there exist headstones in the current project whose textures are displayed improperly.

These headstones pictured in Fig. 20, when measured against their real life counterparts, demonstrate a warping effect with a few visible inconsistencies that are relevant to this role's workload. The first notable thing is that these textures are improperly centered. When given approximate center line recreations shown in Fig. 21, it is evident that the "center" of the engravings in these headstones should line up, roughly, with the highest point of the stone itself, or the top of the

[20] Two sets of headstones rendered in Unity, compared to headstone captured in real life.

stone's curve. The in-engine representation of these stones show that their relative centers do not always follow this guideline, which is an obvious issue. The second effect of this warpage is the "stretching" effect, which is skewing the textured image in such a way that either height or width can be represented incorrectly. This, along with the incorrect center of the image, can cause text to be cut off by the edge of the model.

This is where the importance of accurately mapping the existing tool becomes evident. It is not the team's opinion that the Unity engine itself is warping

these texture files improperly (if it were, likely all of the textures would be affected instead of just some), but rather the tool being used to create the texture that is doing so, which means the texture images that Unity is being provided with are, themselves, incorrectly made. Consequently, the only option, both in terms of this project and in future uses of this tool, would be to fix this problem at its source, which can only exist somewhere in the texturing tool. The only plausible way to do so is to understand the tool, which I have elected to do - line by line, if need be.

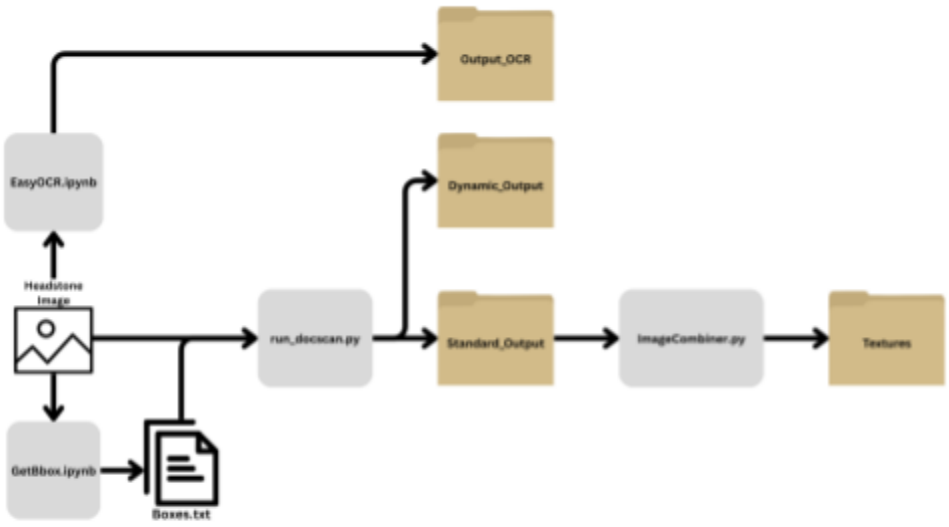
[21] Headstones depicted with approximate center lines.

The second focus of my work in this role, which is aided by the understanding built with the previous step, is the enhancement of the texturing tool, so that it can be used more seamlessly in future implementations by Dr. Giroux. This task requires some judgement calls, where a balance must be struck between scope creep, and ease of use. It can be assumed that, like this project, the person taking the real life photos and feeding them into this texturing tool will not be an experienced python programmer, and it is clear that the current implementation requires some tinkering that such an end user may find frustrating. Because of this, changes and enhancements to the tool that facilitate ease of use can be considered a core aspect of the overarching goal of generalizing it for future use.

The first step in both of these things, of course, is a general overview. Starting at the ingress point, where the raw images are fed into the pipeline, allows for an auditing of all the processes that the image goes through, which helps simultaneously to isolate issues and identify points that could use improvement. Using the DOCSCAN_README txt file in the texture tool, I was able to discern a

flow to how this process takes place. The following flowchart has not been meticulously scrutinized for accuracy, but I felt it was necessary to include it here as part of my preliminary investigation into the tool's inner workings.

As shown in Fig. 22, the headstone images, as far as I can tell, are used in 3 separate places. The GetBbox.ipynb file, where the standard headstones are sent to create bounding box data, the run_docscan.py file, where these images are used in tandem with the bounding boxes (or, in the case of the non-standard headstones, just the image files) to create outputs that go into either the Standard_Output or



[22] Flowchart of how images flow through the texturing tool, file by file.

Dynamic_Output folders, and the EasyOCR.ipynb file, which uses a text-detection algorithm to identify areas of interest. The standard headstones' outputs are sent through the ImageCombiner.py file, which produces the completed textures so that they can be used in Unity.

It should be noted that this flowchart also includes the path for nonstandard headstones, or headstones that are irregular, unique, or otherwise different from the majority of headstones for which the same initial model is constantly re-used.

As such, they are much less of a concern when it comes to the optimization of this tool, given that they require a much more manual touch, and are therefore unaffected by whatever code is causing the incorrect skew on the normal ones. Due to this, the investigation and enhancement of this tool will fall mostly in the realm of optimizing these standard headstones' texture creation. Similarly, because the bounding boxes created by the GetBbox.ipynb file are what get used in the image cutting in files further down in the pipeline, this file is a suitable beginning for investigation. Relating this to my goals of finding the source of the texture warping and improving usability, with this file, as with the ones that will be subsequently researched, I will be trying to answer two questions: "Does it need fixing?", and "Should it be better?".

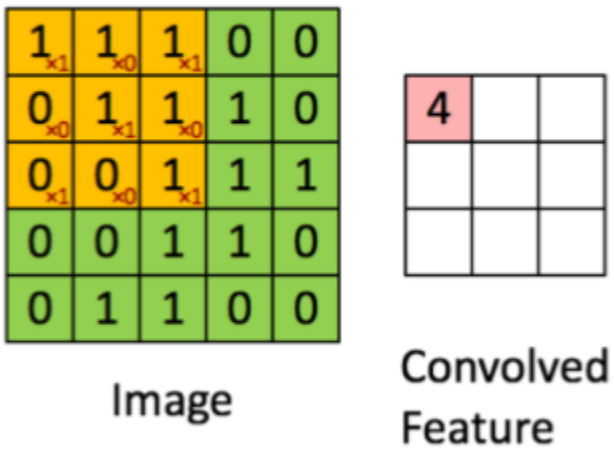
The bulk of the code in GetBbox.ipynb revolves around its usage of Detectron2. This import is a framework created by Meta's AI research division, and it's used for object detection and segmentation. Its claim to fame is variety - it allows for numerous detection algorithms for a variety of needs, and it is the primary tool that this project's texturing tool uses to create its bounding boxes for this project. The subject of this tool's segmentation is the headstone itself. Detectron2 looks to isolate the actual shape of the headstone so that, later in the pipeline, a cut/crop can be performed on the image that produces an output whose dimensions are exactly that of the stone. This, ideally, would preserve the aspect ratio of the original image.

There is a possible ethical issue which arises from the conflict of mixing neural network (hereby abbreviated as NN) procedures with real life. There exists a somewhat inescapable idea of error within the operation of a NN; they cannot operate without loss - so much so that there exist functions that operate on the assumption that this error always exists ("error" in this context meaning any output that is either misclassified based on headstone type, or inaccurate in terms

of the bounds of the object detected). This concept puts itself at odds with a potential issue of respect. Put simply, it would be disrespectful to say that a project is satisfactory, or in any way finished, when included in that project are headstones of real life soldiers that are depicted with a drastic skew, or letters cut off from the side, or in an otherwise unpresentable manner. Thus the output of Detectron2 must be met with an extra layer of scrutiny. That's not to say the very concept of errors cannot exist, however. This is merely a vehicle to drive home the point that errors must be kept to an absolute minimum, more so than they might under a different project.

This tool uses a custom config .yaml file to specify the use of one of Detectron2's included detection algorithms. It makes reference to a Generalized RCNN Meta Architecture, which is a Region-based Convolutional Neural Network (acronymized RCNN). The "Generalized" adjective simply denotes that it is one of the RCNN initiatives offered by the Detectron2 library, where the specific version used will be specified later. A key aspect to the function of a RCNN is the idea that it warps images (among other things) to aid in its prediction computations. It may seem like an obvious first thought that this could cause the warping issues we are seeing in the output - an idea which necessitates an understanding of this predictor, in order to decide whether or not this causes any harm.

A convolution, as far as its use in neural networks is concerned, is a process that takes a "kernel" of a given size (3x3, for example), and applies it to certain windows in an image, with each pixel inhabiting a cell that is taken into

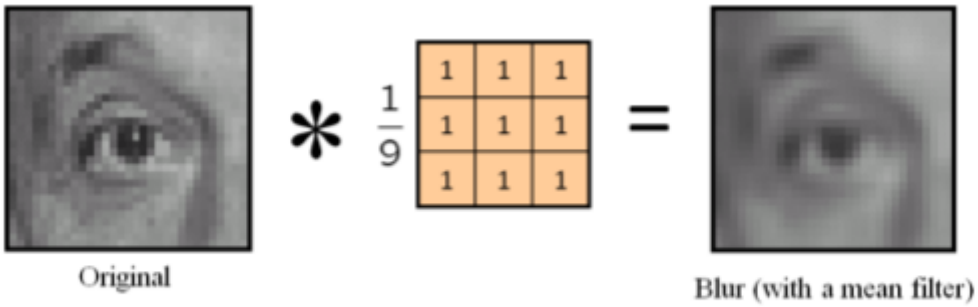


[23] A convolution with a 3x3 kernel (yellow, with values in red text) overlaid on a source image matrix (in green), with output depicted on the right (in red). Image from Nvidia Corporation.

account by the kernel. As shown in Fig. 23, this produces an output of a certain value, which without context may seem arbitrary. However, if this convolution were iterated over every possible 3x3 window in this image, it would fill the 3x3 output on the right with those calculated values, and we would call this output a “filtered” image, or an image with a calculated filter that has acted over it.

Pictured in Fig. 24, it’s evident that an image can have real life effects when acted upon by a kernel. These effects are possible because each pixel of the output image exists because it was produced by also taking into account all the pixels around it. It is this behavior that allows it to be used in something like edge detection, where stark changes in pixel color values can be detected by a kernel.

This makes it a valuable tool in the field of neural networks, and is the

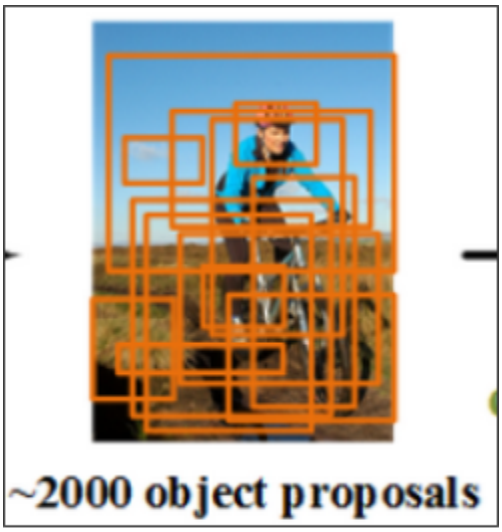


[24] An image that is filtered with a blurring kernel. Image from Cornell University.

primary calculation done by CNNs. The purpose of this CNN is simply to decide where the primary object is, and where it isn’t, using this detection kernel. The general drawback of sliding window CNNs is that entire subjects are often too big to be entirely captured by the Kernel. Since neural networks are computationally expensive, multiple kernels of varying sizes quickly becomes an undesirable solution, and it adds to this already expensive workload quite quickly.

This is where RCNNs make an impressive improvement to the original CNN formula. The Regional-base (The R in RCNN) involves a new concept called region

proposals. Region proposals utilize a selective search algorithm over the whole of the image in order to produce a number of “regions of interest” that can be used for convolutions later in the pipeline. These regions (around 2,000 of them) are generated with a high yield, low accuracy mindset [citation] that means there will be a high amount of overlapping segments (pictured in Fig. 25) that are considered “of interest” to the algorithm. Regions can be looked at by the CNN to help identify objects, without having to take in other parts of the image, which would be unnecessary.



[25]An example of several overlapping sections in the region proposal algorithm. Image from Tsinghua University.

Using an algorithm called Non-Maximum Suppression, these regions can be iterated on and refined to remove overlaps and constraint the region to being as tight around the main object as possible. This process, in effect, makes the RCNN a many-branched network simultaneously working to identify multiple objects within one image, depending on the amount of objects that it is configured for identifying.

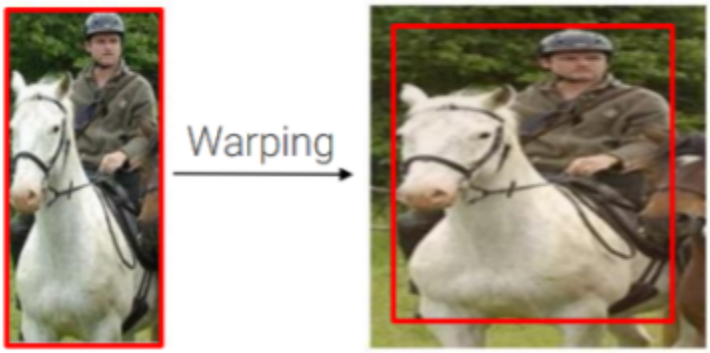
RCNN comes with a considerable downside, however. This region-splitting process takes time, especially when considering the number of different objects in an image that may require classification. These regions may be acted upon by the CNN portion of the network before they are refined, which leads to recomputation of the same section of pixels by multiple regions’ convolution. Which begs the question; in what way does this texture tool benefit from - and therefore necessitate the added complexity of - an RCNN? The answer to that question lies in the output of Detectron2 itself. Its function in this file is in two goals -

establishing a bounding box for the image files given, and identification of standard vs non-standard headstones. (Note: there is a “Grass” headstone category that retains many aspects of a standard headstone’s pipeline journey). There is enough variation/complexity within the standard headstones themselves to, in my opinion, warrant the use of an RCNN over a standard CNN. The region proposal structure of RCNN adds a layer of certainty that the whole of the headstone would always end up being detected, and therefore computed by the CNN network, preventing the more unique details like text or decorative embossing to falsely register the image as containing a unique headstone, when it otherwise would be correctly labeled as standard.

Scrubbing through the config.yaml file for the GetBBox.ipynb tool also provides some context that is useful in answering this question. It was mentioned earlier that the meta-architecture for this implementation of Detectron2 is a “Generalized” RCNN, and that the specific version would be specified elsewhere. The yaml file includes settings for the RoI (Region of Interest) Heads, which include the specific RCNN algorithm that is being used: Fast RCNN. This version of RCNN reorders the usage of the CNN portion in order to reduce redundant calculations. First, the CNN is run on the whole image, rather than each individual region, and the Region Proposal is used with this CNN output in something called a RoI pool. Each feature map from the CNN is checked against the corresponding regions for a fixed sized feature extraction, which is passed into the fully connected CNN, and the bounding box refinement processes mentioned before. With the initial layers of the CNN not being fully connected, a sufficient number of “good enough” outputs are produced that region proposals can be used only so much that the maximum benefit can be extracted from their inclusion, while reducing their impact on the computations as a whole.

All of this works to answer one of the principle questions: should it need to be changed? Given the need to be able to reliably classify headstone images by unique vs standard, RCNN architecture is justified. Similarly, given the more than six hundred headstones present in the cemetery that need to be turned into textures for the Unity project, the speed of the Fast RCNN algorithm alleviates the runtime and memory downsides that traditional RCNN introduces. In short, I would not say that this file should (or substantially could within the project scope) be improved for future use.

The other main question still remains: does it need to be fixed? It was mentioned earlier that a CNN warps the image for the purposes of computation. This was something I learned about in the beginning of my research, and made sure to keep tabs on as I researched Detectron2's implementation, in case this ended up being a root cause of the warping. Definitively, and perhaps, obviously, this is not the cause of the warping issues in the Unity project. In any CNN, the window in question is warped to match a specific size, one that matches the aspect ratio of the convolution kernel. Fig. 26 shows one such example. All of this warping is merely done for the purpose of compatibility with the Convolutional layer. It is temporary, and thus has no effect on the output of the image.



[26] An Example of an image warp performed by a CNN for computation. Image from University of Berkeley.

Furthermore, after viewing the output of one of these files, pictured in Fig. 27, it is clear to me that the output of these files appears correct. This particular headstone is one that experienced warping issues in Unity, and is one of the ones

pictured previously in Fig. 20. Despite that, the output obtained by Detectron2's Fast RCNN algorithm is clearly correct in its classification, acceptable in the computation of its bounding box, and, most importantly, free of any warps. So, as an answer to that question, I would not say that the GetBbox.ipynb is likely not the root cause of the texture warping issue, and is thus not in need of any fixes.

[27] An output file obtained by Detectron2's visualizer.

Another file that deals with outputting information relating to the headstone images is the EasyOCR.py file. EasyOCR is a neural network model maintained by Jaided AI and implemented with the PyTorch library, which itself is a popular neural network library that specializes in modularity and ease of use. Optical Character Recognition (OCR) is a decades old framework, whose robust nature ensures it's still employed by modern day algorithms, like the EasyOCR being used by this project. It is made for the sole purpose of text classification, it can read characters with a high degree of certainty, with a goal of maintaining accuracy in various fonts, text orientations, and photo conditions.

The first thing that's done to an image being passed into EasyOCR is preprocessing. An image is made to be grayscale, to reduce confusion between color values. Images are also thresholded, which is the process of skewing these grayscale pixels to their closest extremes. The effect that this has is that rapid changes in color values within the original image are amplified. This amplification makes those changes more "obvious" to edge/object detection algorithms that act

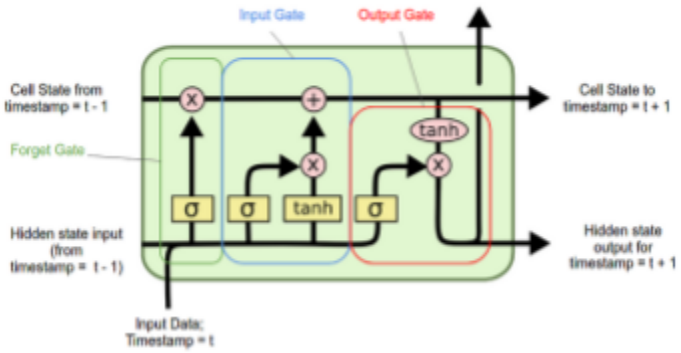
upon the image. An example of this is shown in Fig. 28. This works to the benefit of things like CNNs.

[28] An image that has gone through a thresholding process. Image from GeeksforGeeks.org.

Another protocol at work within EasyOCR is ResNet, which is a kind of CNN, like discussed with Detectron2. ResNet stands for Residual Network, and it refers to a modification made to typical neural networks. In normal operation, a CNN network, throughout its many layers, may stray quite far from the original essence of the object it's looking at. This can be a difficult idea to conceptualize, but in my research I came to understand it as a sort of "telephone game" consequence of repeated calculations upon an already changed output. This makes sense when considering that CNNs must, in their convolution, manipulate the image to fit the dimensions of their kernels. Further and further iterations on a manipulated image can cause a loss of detail. This concept, while not limited to visual-based neural networks, is often referred to as the "vanishing gradient". ResNets alleviate this contextual issue by supplementing the original input back into the CNN, at some stages of its computation.

Easy OCR also utilizes LSTM algorithms, which stands for Long Short-Term Memory, which is a type of Recurrent Neural Network (RNN). As the name implies, Recurrent neural networks make use of recurrence in their computation, where the input of a computation node is the output of a previous iteration of itself, plus a modifying value. Recurrence makes its usefulness evident through its assistance with contextual problems, like language. Technologies like predictive text require context in order to predict what will next be given, and recurrence allows the node to take in previous data, to decide on what context to use. If a program needed to figure out whether a “wh” would turn into “why” or “where”, it could look back at the previous words to decide.

LTSM is a modern implementation of an RNN that exists to combat an existing problem with typical unchecked recurrence; complexity growth. Using language as an example again, a long chain of words or characters will require additional computation to gather context for, as predictions and inputs grow. This very obviously has an effect on training and running time. An LSTM fixes this by adding an interesting new subnode to the traditional RNN node; a “state” cell. The state cell (Pictured in Fig. 29) includes three gates: input, output, and forget. The input gate governs what may want to be “remembered” by the cell. Things in language, like tense, or plurality, could be important to context, in the prediction



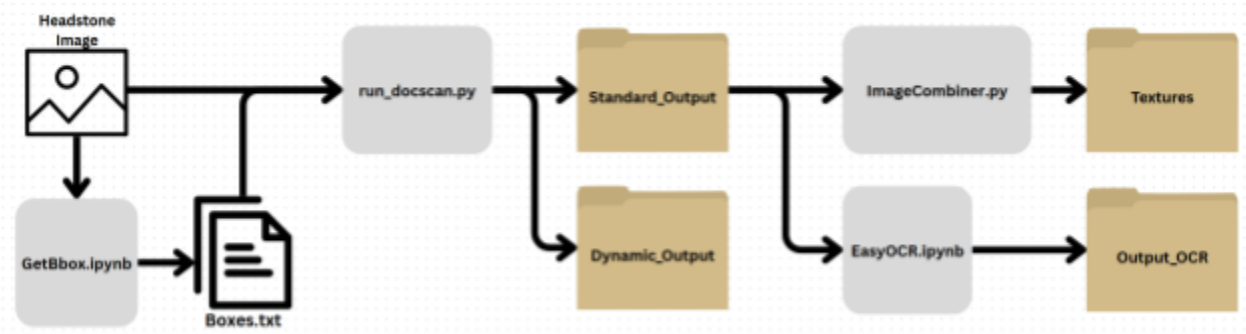
[29] A State cell in an LSTM Node, with forget gate (Left), Input gate (middle) and Output gate (right). Their functions are depicted within the cell. Image from Analytics Vidhya.

of future words or characters. If these should find reason to be used, they are put into use by the output gate. If something changes, like plurality going from singular to multiple, then the previous state of that gate may be discarded by the cell with the forget gate. This usefulness is enhanced by the idea that these changes happen in gradients, or weights. If the node decides that plurality may come up in the future, it may not want to completely forget it, and thus the usefulness of it is preserved. By the addition of this new state, the next node may be able to derive all the context it needs right at its instantiation, with little need for useless backtracking. This has an appreciable effect on runtime, and is an invaluable tool in the EasyOCR network.

Another key tool exhibited in EasyOCR operation is Connectionist Temporal Classification (CTC). CTC addresses a particular issue within text detection workflows; how does one ensure the program separates words correctly, or keeps a single together when it would otherwise be separated? To understand how a CTC can help with this issue, I must understand the concept of "loss" when referring to neural networks and deep learning algorithms. Put simply - loss is a way to describe the deviation from the network's output to the "ground truth", or the objectively correct classification that is already known. If a network correctly labels 98% of a given set of inputs, then a loss function would describe that 2% that were mislabeled. CTC is one such function. This loss function is taken into account whenever the EasyOCR network encounters abnormal alignment or spacing between characters. The headstones within the cemetery feature a large amount of variation in fonts, orientations, and letter spacing, so it's easy to see the benefit that this function provides. It does so by appending strategic "blank" symbols within areas of uncertainty. These symbols act as markers to the network, which can verify these symbols against its training data, allowing it to make informed decisions on what to segment, and what to keep together. CTC functions excel in "Many to one" scenarios, where a number of different elements would

ideally point to one “correct” answer. OCR is one such scenario, which justifies its usage in the EasyOCR pipeline.

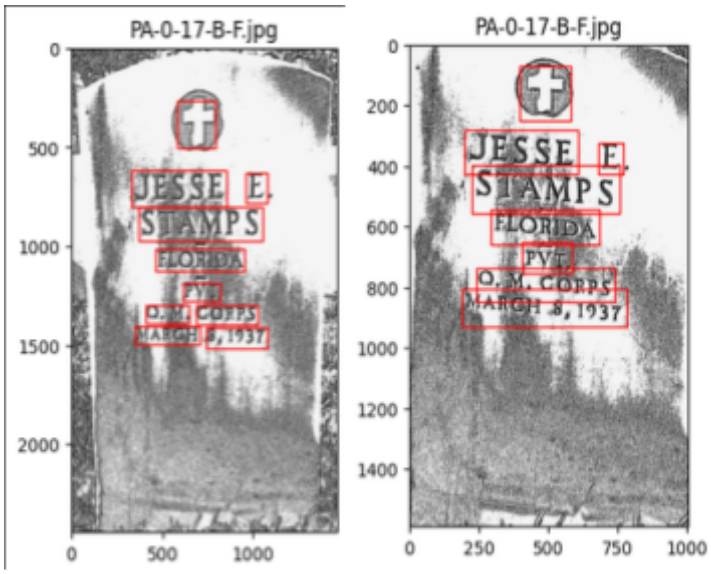
There are a number of other smaller features within the EasyOCR pipeline, but these are the ones that make up the bulk of its functionality. As such, its usage in the EasyOCR.ipynb file within this project’s texturing tool is somewhat simple. The image is first pre-processed with python cv2 functions. It is gray-equalized, then gamma corrected, and then tone corrected, and finally thresholded. Using an imported dictionary, the EasyOCR tool is imported and run on the given image set, and then the output is saved to a text file that holds the detected text on the headstones. An important distinction is to say that this tool is run on flattened, cropped images. That is to say, the intended input is a set of images that have gone through run_docscan.py procedures. This means two things: firstly, this is when I realized that my initial flowchart is incorrect. Where I previously showed in Fig. 22 that the images put through the EasyOCR pipeline are the original, unedited versions, I now know that they are actually from the Standard_Output folder (or at least a similar copy of them). The corrected flow chart would appear as follows, in Fig. 30:



[30] The corrected flowchart, where images from Standard_Output are used as input for EasyOCR.ipynb, rather than unedited headstone images.

Secondly, It means that, in my opinion, the text output by EasyOCR.ipynb with Standard_Output as an input is more accurate than text output by the file

where the original images are used as input. From a far away view, I believe this makes sense; the images in Standard_Output are ones that have been through the run_docscan.py process, which itself calls several other files, one of which operates upon the images with multiple cv2 image manipulation functions. These functions, found in docscan_args.py, include dilations, contour draws, perspective transformations, and more. As these are all done in the service of making these headstone images appear more clear, and "straight on" from the viewer's perspective, it stands to reason that they may also have an appreciable effect on how text gets detected by EasyOCR. Fig. 31 depicts one such example. Clearly, the left image is zoomed out further, and the text blends in more with the shadows cast on the stone than that of the right picture (notice the "I" in "Florida"). Similarly, we see a difference in segmentation of object detection boxes between the two. Where the output on the right reads the "Q.M. CORPS" as one instance of text, the output on the left views them as separate entities. The same happens with the obituary date one line lower.



[31] Two images of the same headstone, run through the EasyOCR network. One is of the unedited headstone picture (left) and the other is of the edited picture (right).

Knowing these things, a question arises; does this account for any differences in the text that is output by EasyOCR? Looking at the text that is detected with the unedited files as input, we see two things in the PA-0-17-B.txt file - Those separated boxes, due to the file structure defined in EasyOCR.ipynb, mean that would appear on separate lines in the output, where they would

otherwise be on the same line. The other noticeable thing in this text output is that “Q.M.” is detected as being “O.M;” and “1937” is detected as being “4937”.

At this time, it is unknown whether or not these specific errors would be included in the case of the cropped and pre-processed images being used as input. It is clear that a difference can be discerned with the human eye, but it is hard to know whether or not the errors would still be present. The image on the right in Fig. 31 is a demo preview left in by another user, so there is no output folder to look at, at this time. This will be the subject of further investigation, and this paragraph will be modified to describe that process, when it happens.

Of course, this analysis must be contextualized to the two questions I asked of GetBbox.ipynb. Does EasyOCR.ipynb need to be fixed? Relating to the warping issues - almost certainly, no. Image Fig. 20 makes clear that any stretching or skewing that happens in this output is of another program’s design. More importantly, the output of this file is not a component of the texturization process of the images, at all. It merely latches onto the midpoint of it for its own use.

Should it be improved? My initial assessment of the tool indicated that there was work to be done for the purposes of guaranteeing the most accurate output possible for use in the queries that I had surmised were the intended use of this data. Truly, if there were any reason to use the data from this file, it would be in the context of providing information to the VR user, which would necessitate accuracy of the EasyOCR tool’s output.

This, as I learned, was of no consequence, in the end. After some further investigation, Dr. Giroux, the sponsor for this project, informed me of this tool’s uselessness in the current build. There was already a database of information available for input into the Unity build, and thus that work has already been done. As it happens, this tool was, as described by Dr. Giroux, a sort of unwanted detour

by one of the previous students, and none of its outputs are in use anywhere in the final build, and that is not likely to change. This file is, for the purposes of my investigation, a dud.

It does provide a passing value in its presence, though; It was this file, and the output pictured in Fig. 31, that acted as my moment of clarity for where specifically those textures may be encountering warping issues. It was here that I first noticed a difference in output characteristics where input images were taken from different sources. To contextualize this, we can continue to follow the pipeline after the Detectron2 output is generated; turning my focus to GetBbox.ipynb, the question becomes: what happens to the output of these files? The GetBbox.ipynb file indicates a well-defined structure to the boxes.txt file that it creates. Each line, which itself describes a single image, includes a name adhering to a schema that allows for easy relation with the original image. This is followed by x, y, height, and width values for offsets and bounds, respectively. The line ends with a single digit, 1, 2, or 3, to indicate standard, grass, or unique headstone types.

This text file, along with the original batch of images, are then fed into the next file, run_docscan.py. The first thing that this file does is call another one, check_boxes.py. This file is passed the data from boxes.txt, and checks if the height of the computed bounding box is greater than the width, which is true for all standard headstones. If this threshold is not met, the program makes sure that it is classified of either type Grass, or Unique (2 or 3). Answering my previous questions is a relatively simple task with this file; it exists as a dummy check incase of specific errors with Detectron2's output, and does not, in my opinion, necessitate specific "ease of use" improvement at this time, as the user of this tool does not interact with it. Similarly, I have verified that the textures being improperly warped in Unity are not ones that have been affected by this file, which

means I also do not expect that it needs fixes. After that file finishes, the remaining code in run_docscan.py can be performed.

What happens next in this file is a point of major importance in the investigation of this tool's shortcomings. My analysis into this is built upon hypothesis, for the sole purpose of aiding Garth Simpson in his more granular search of the warping problem's source. As such, I will touch upon topics in this section that will be explained in more detail later.

The next major thing that happens in the run_doscan.py file is that a subprocess is called on all of the generated outputs generated by Detectron2. Docscan_args.py is a program that takes in the bounding box information, relates that info back to the original image, and then performs a number of operations on it. In some form, all of the images, standard or not, pass through this file for manipulation, but it is the standard headstones that receive the most manipulating, in this file. As such, it is my suspicion that the number of operations happening to a standard Detectron2 output, paired with the "unsupervised" nature of said headstone's generation, may be resulting in warping.

OpenCV (abbreviated as "cv2" going forward to align with its program usage), is a python library that contains many operations that can manipulate an image. It's quite popular, so its usage in this file is unsurprising, and - as a concept - completely acceptable. It manipulates the images in the desired fashion by using the aforementioned kernels, like in Detectron2, to change a pixel based on its relation to its neighbors. It sees heavy usage in the docscan_args.py file - which speaks to its usefulness, as hand coding any of these procedures would be both time consuming and inferior to cv2's library.

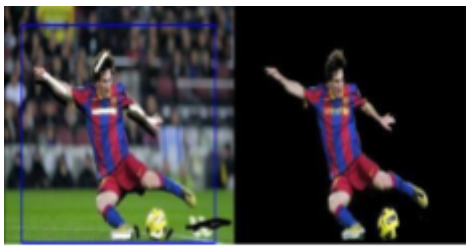
These operations do have implications as to how our textures are generated, however. And, with the knowledge of the bounding boxes existing as "starting

points" for this file to perform those operations on, I can trace through these operations and identify potential issues. What follows is a list of cv2 operations or functions that use them) that I believe warrant testing to ensure accuracy:

- Image Closing
    - One of the first operations to act upon the image. The background is blurred and the text is partially (temporarily) erased in order to aid edge detection later on. This is done with cv2's image closing (pictured in Fig. 32). Should further manipulation rely heavily on this output, errors could amplify visual bugs throughout multiple operations.

[32] Image closure. Image from OpenCV Documentation.

- GrabCut
    - Returns a "cut" output of an object detected in an image using GrabCut (pictured in Fig. 33). Similarly to before, errors present in output of this function may cascade into larger and more obvious errors later.



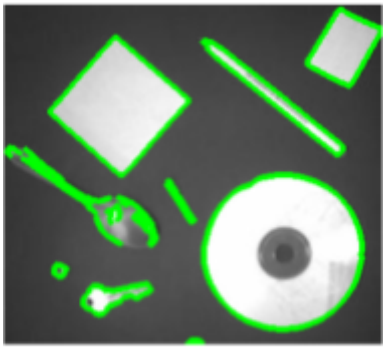
[33] Example of a cv2 GrabCut. Image from OpenCV Documentation.

- Edge
    - Runs a cv2 edge detection algorithm (pictured in Fig. 34) that both grayscales and blurs the image before running a canny operation, followed by a dilation. The canny

[34] cv2 canny edge detection. Image from OpenCV Documentation.

operation in particular presents a risk in the case of errors.

- Contour
    - Runs a contour detection on the image for the purposes of accurately defining a shape. This function returns only the largest detected contour (detected by the function described in Fig. 35), presumably in terms of area. If the correct shape of the headstone were detected incorrectly, a different area could be returned by this function, which may present problems.



[35] cv2 contour detection extracting general shapes of objects. Image from OpenCV Documentation.

- Perspective
    - This function (pictured in Fig. 36) lies much further down the pipeline than the other functions, but performs a similarly drastic operation that manipulates the perspective of the image based on a getPerspectiveTransform cv2 operation, and a warpPerspective operation. The first is used in the computation of the latter, so either of them producing an error could have an appreciable effect on how the image looks.

[36] cv2 perspective shift changing the perceived direction of where the sign is facing. Image from OpenCV Documentation.

These five functions stick out to me as potentially dangerous, due to both the reliance on cv2 to detect hundreds of different headstones in the way that we want, and their influence on the image manipulation as a whole. Placing such importance on elements that introduce a degree of uncertainty means that the factors governing their operation must be strict, or else potential problems may magnify as the image manipulation progresses. It is with that mindset that I make the hypothesis that this file could very well be the source of the warped images in the final build of this project.

To illustrate the effect they have on the headstones specifically, I put together a small testing file to run a specific headstone image through. This file currently performs four of the five major operations that happen to a standard headstone image in docscan_args.py. These operations (pictured in Fig. 37), all operate on the output of the previous operation, which confirms the potential danger that errors could cause. The further back that the error lies, the worse that its effect may be on future operations that rely on correctness to produce stable output.

As a quick side note, the cv2-closed headstone may appear unchanged at the zoom level exhibited in this document. Fig. 38 shows a closer look at this output, depicting a blocky appearance. This operation has the effect of removing small details, so while preserving large color changes. This is to enhance the influence

that the edges of the tombstone have against its differently-colored background.

[37] A picture of a headstone that has gone through three major cv2 operations: closing, canny edge detection, and contour detection, respectively.

[38] A close-up of the headstone that has run through the Canny operation

This, in and of itself, doesn't present any glaring issues, in my opinion. Turning to its usage within the Canny edge detection, however, we can glean some interesting details. Fig. 39 shows a place where a contextual error is present. The side of the headstone, in the area where a similarly colored-fence sits behind it, failed to have that portion of the edge detected. This is a clear indication that similarly colored-areas present a possible issue, with canny edge detection in particular. I have two major thoughts about this finding, the first being that

real-life scenarios make this an extremely tricky thing to avoid. Many places in the cemetery are likely to include conditions that facilitate this error. So, while this could, theoretically, be the fault of cv2's closing algorithm bringing those two areas too close together in color value, the too many variables exist elsewhere to make trying to tweak parts of this closing algorithm feasible. Indeed, my investigation into that point makes clear that this behavior also occurs when closing is not before the edge detection algorithm.

[39] close-up of a canny edge detection output pictured next to its input

My second thought is that these things may be a contributing factor to the overall bugs in this tool. We can tell immediately that the contour detection run on canny's output does not fix the issue of these lines being missing from the photo. Therefore, it may be important that the functions that take place after this accurately account for these errors, lest they contribute to warping.

It's at this point where such scrutiny tapered off, from my end. My primary goal in the role of "texture team" has been to make a judgement on the AI models themselves, along with whatever files immediately surround them. I am responsible for these tools in service to their implementation within a new prototype, as well as making judgement calls pertaining to their viability. I still have a vested interest in finding the source of these warping issues, but with the files that we planned to be under my direct purview likely being a non-issue, my role in that investigation takes on more of an advisory nature, as Garth provides a more in-depth look into these functions. Thus, further discussion of docscan_args.py's potential issues will be described soon.

With a large amount of research out of the way, I would describe that workload as such:

- Research on methods of consolidating the actual operation of the tool (Where and when can scripts/batch files be used in order to cut down on manual configuration by the end user?).
- Consolidation of these tools into a central location, utilizing the new texture tool prototype with Garth Simpson.

This “consolidation” is done with two goals in mind: ease of use, and better accuracy. As it stands, multiple programs included in the existing tool include instructions on their use. These instructions often dictate that the user copy the tool to their drive, running that tool from the copy, and designating folders for varying outputs. Given that part of this tool’s goal is to be generalizable, and easily usable in the future, it is in the best interest of the project to streamline the process when possible and appropriate. This new prototype, built in a Google Colab file, is meant to be that. The workload for implementing it is split up between Garth and I, which means I will be responsible for implementing only those functions that I have personally researched, and deemed necessary for this tool’s future use.

One obvious thing that this implies is that EasyOCR will simply not be present in this version of the tool. While there is a possibility that other cemeteries that this tool may see use with will not provide Dr. Giroux with a list of headstone information as has happened this time, it is my opinion that the use of EasyOCR and the surrounding real-life things that can affect its performance, are not worth the effort needed to verify accuracy. As it currently exists, I would not trust this tool to accurately detect the correct text, and would feel the need to personally look through every file for accuracy, which defeats the point of using it in the first place.

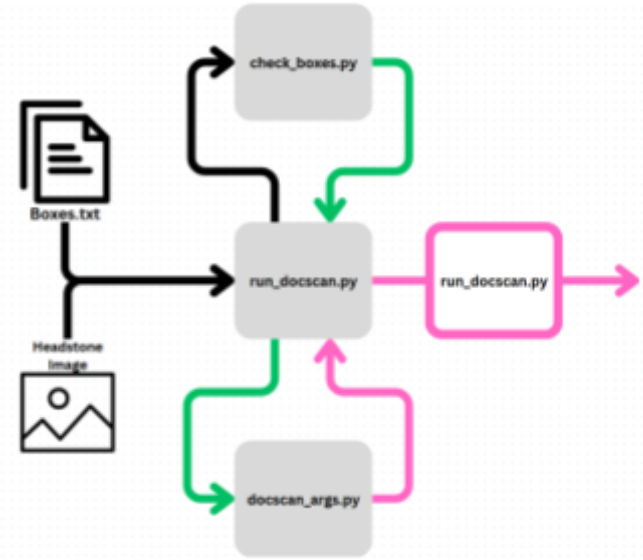
Detectron2, however, is another story. It will continue to see use in this project, though likely not as directly as other functions would; I can foresee cause for its logical separation from the prototype file itself, even though it would be used at least once in different iterations of this project. Given its runtime, it would be disadvantageous to require that it go through these neural network procedures every time the prototype is run, in the case that Detectron2 has already produced usable output. It would be much better if the rest of the program could be run quickly as many times as needed, with the GetBbox.ipynb files only being run once. So, much like the current tool, the GetBbox.ipynb file will exist in much the same way, in relation to the new prototype. Having only one extra file to worry about would be a much more preferable solution to the way that the texture tool exists now.

Additionally, Google Colab's markdown structure allows for a more modular approach to the way that these features are added. With proper separation of code, certain parts can be formatted into cells in such a way that one can run only the cells that they want to, with the caveat that those cells have all of their values instantiated and available for use (this last point is why I still insist on the separation of Detectron2 from the main prototype file. Were the runtime to be disconnected, the cells containing Detectron2's output may need to be run again when they otherwise wouldn't have been). This, in my estimation, makes for a simpler structure to read through, provided the inclusion of "clean" code and instructions for non-programmer users.

This means that a portion of the existing tool's code must be "translated" into the prototype, which involves some rewriting. There are a number of places where the code has to designate areas of output within Google Drive, for the purposes of retrieving this same information later in the same tool. Inherent to the separation of these programs is a retrieval stage for each of these (In the case of

GetBbox.ipynb, there is also a saving stage). The docscan procedures have a retrieval process such that two other files, check_boxes.py and docscan_args.py, are run as subprocesses, which is useful in that it doesn't cause unnecessary "bloat" on the Google Drive where other outputs exist. This behavior can be brought into the prototype in an identical fashion. However, due to the decision to have the Detectron2 procedure be separate from the prototype, those behaviors will have to be kept, meaning boxes.txt will continue to exist as a text file.

This means that there will be a short "unpacking" phase of the prototype, where it retrieves its input, the boxes.txt file, as an already completed entity. This currently happens in a (very small) hub-like structure, where the run_docscan.py program acts as an oversee, detecting whether or not the texture tool is called with the expectation that Detectron2 will be run with it or not (this is a topic that will be discussed shortly). It manages the retrieval of the images, performing its double-check with the previously described check_boxes.py, and then sending the images through docscan_args.py. Once this process is complete, it passes on to the rest of the pipeline as part of either the "Standard Output" folder, or the "Output" folder. Pictured in Fig. 40 is an illustration of this concept.



[40] Flowchart illustration of how information flows through run_docscan.py.

As mentioned, there is, in the existing tool, an option to run these files without using Detectron2, and thus without boxes.txt as an input. The processes, and number of them, that this goes through are different, but for the purposes of texture automation, they are

irrelevant. This process is done for unique headstones, and texturizes them in a different way. As far as the prototype is concerned, this functionality will not see use in that specific file, though the potential need for it for future use has been acknowledged.

With all of that, I've arrived at a tentative roadmap for the prototype, and the basis of my work going forward. This will include what I expect to be necessary for my role in the texture automation, and though I expect my work will include supplementary assistance for my teammates in their respective roles, this portion of the project is what I am responsible for. This work includes my portion of the prototype, which will be built with the following roadmap in mind:

- The Detectron2 procedures will be separate. Their input will remain the same, as will the structure of its output. Its output will be called into the prototype for reusability purposes.
- EasyOCR will be kept out of this, as it provides no use.
- The prototype will concern itself with the standard headstones, as those are the outputs that are meant to be good enough to use without having to tweak them.
- The bounding box info will be fed into a simple array, as it does not need to work in real time, so complex data structures are not an issue. This array will be built to fit the needs of the procedures Garth will use in the docscan methods.

That last point may see some fluctuation. It has been discussed that it could be useful to Garth for me to also export an extra x and y value to boxes.txt, acting as corners, so that he can avoid having to calculate those in the program further down the pipeline. Whether or not this will come to pass is a small matter, but I

speculate that it may aid in avoiding warping issues, should he decide to use different cv2 procedures.

I will also be engaging in more rigorous testing that aligns with the spirit of my research. This will help to identify potential solutions, as Garth and I will be able to cross reference the same information, but in the context of our respective areas of research. My testing for warping issues will include the following targets:

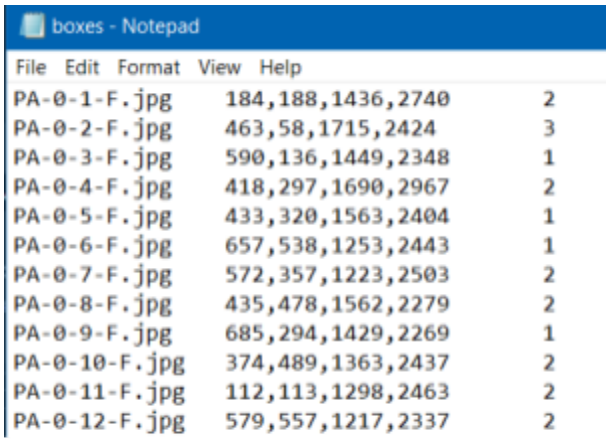
- The five aforementioned methods in docscan_args.py, for their "output is the next method's input" behavior, to verify whether any errors cascade as they travel down the pipeline.
- Bounding box buffers and static value tweaks, to act as a possible safeguard against whatever error Detectron2 produces in its output.
- Other cv2 functions that could act as error-checks for issues in previous outputs (Is there anything that can act as a band-aid for canny's flaw?).

This testing will act upon images that are known to have issues in the final build, so that fixes can be more obvious and direct.

State of the Local Python Files

After exploring the local files of the St. Augustine Cemetery project, the methodologies for flattening the headstone images have been identified. The files require a "boxes.txt" file as well as the tombstone images. A majority of the texture flattening algorithm occurs in "docscan_args.py," and the other files are used to support it.

The output boxes.txt has the bounding boxes for each headstone. The file has the name of the image, the pixel coordinates for the bounding box, and the type of headstone. When these files are flattened, they are run on three cases:



[41] Bound box text file

1. The headstone is a standard rectangular shape and is not lying down.
2. The headstone is lying in the grass
3. The headstone has a unique shape

The most challenging case for automatic texturing would be the uniquely shaped headstone. To flatten the face of a headstone, the program needs to find the four corners that estimate the perspective of the headstone. However, headstones in case three have a large array of corners and organic shapes that can cause many headaches. According to Dr.Giroux, it has been determined that unique headstones do not need to run under an automatic texturing algorithm. In the scenario where unique headstone images could be flattened, it would be difficult to approximate how tall or long the flattened image should be. Stretched images would appear stretched in the VR scene, but it would also cause a lot of unintentional cropping. Therefore, the project will focus on making automatic texturing for rectangular-shaped forms. Any tombstone in case three will be

manually textured using software such as Substance Painter or any image editing software.

In total, the automated texturing system will use 594 provided photos for the project. Some of these photos include the back of the tombstones if there is any unique engraving. All the other headstones will have a generic stone texture applied.

[42] Tombstone types 1,2, and 3, respectfully.

The local files are run through the “run_doscan.py”, and it requires the headstone photos to run. The project is capable of running without the “boxes.txt,” but the results can vary as there is no way to separate cases between different tombstone shapes. If a headstone does not belong to case one, a standard headstone, they are run over a dynamic detector. “run_doscan.py” is the file that

handles these cases by looking at the very last number of the line of the “boxes.txt” file.

Function of the local files

The other locally running Python files are:

1. The “Docscan_args.py” file has almost all the functions for completing image manipulation for flattening tombstone photos. The process works by creating or using the bounding box in “boxes.txt” to crop the photos before running edge detection on it. To prepare for edge detection, the background content and text are blurred. Contour detection is run on the result of edge detection to find the edges of the headstone. Contours are needed to create the four corners for perspective warping.
    a. find_lines(con)
        i. Finds straight lines of the headstone given an image of the subject’s contours.
    b. m_close(img)
        i. Removes text and background formation by blurring so the image can be run through edge detection. It returns the blurred image as a returned value.
    c. m_cut(img, rectangle)
        i. Masks out the background elements from the image and returns the resulting image.
    d. edge(img)

    i. The function uses Canny to find edges and returns the result directly from Canny.

e. contour(img,canny,num,num_contours)

    i. It uses the edges found from Canny, the original image, and the number of contours, and finds the largest contour from the edges. The function returns two values, the image with the drawn contours and the main contour of the photo.

f. corners(img, page, num)

    i. The function loops over the contours and finds the four corners of the contours. It uses order_points(pts) to organize the corners by location and returns the list of corners.

g. order_points(pts)

    i. It organizes a list of given points by location, sorted from top-left to top-right to bottom-left.

h. get_coords(corners)

    i. Returns the final destination of the corner coordinates given a 2D array of coordinates. The coordinates returned are meant to turn the tombstone into a flattened version using the maximum width and height of the corners.

i. perspective(img, obj_corners, dest_corners)

    i. Returns the flattened image of the headstone after perspective transformations are completed. The flattening uses OpenCV's getPerspectiveTransformation and warpPerspective functions to get the final result. The warped image is then returned.

2. “Check_boxes.py” changes the “boxes.txt” file so wide headstones are labeled as a type 3 headstone.
3. “ImageCombiner.py”
    a. find_overaly_images (directory)
        i. This function creates an array with the front and right image files. Given a folder directory, it will loop through each file and look for similar numbered images that contain an “r” or an “f”. Images with an “r” indicate the photo captures the rear of the headstone, while the other letter is for the front. The images are combined into a 2D array for use in the combine_images function.
    b. combine_images(background_path, overlay_images_info, output_directory, positions_and_sizes, resolution)
        i. This function combines the photos of the headstones with a basic stone texture. The photos of the front and back of each headstone are combined in a single image and placed onto a stone texture. The output is a folder with all the textures.

The local files rely on Python’s OpenCV library, which is made for computer vision, machine learning, and image manipulation. Google Colab has a different library for OpenCV, which means code downloaded to run locally cannot. For example, cv2.imshow() needs to be imported as cv_imshow() on Google Colab to work. OpenCV also has its methods for object detection, but without the need for training an AI model.

There are no plans to use libraries and AI models outside of the ones used in the main project. The current state of the automated texturing indicates the

image manipulation needs fine-tuning to run as intended. There may be cases where the previous group manually textured standard headstones because their code was unable to flatten them properly. If a similar problem arises, solutions from the automated texturing prototype will be useful.

Research on Contour and edge detection

To extract information from photos, there must be some way to simplify. As long as objects in an image have some sort of contrast, an edge can be extracted from them. Contrast and edges can be created by:

- Color differences
- Lighting and value differences
- Depth changes
- Surface texture

However, images might have too many points of contrast or sudden changes between blocks of pixels. When using edge detection to reduce extraneous details and noise, Gaussian filters and converting the image to grayscale are required. Multiple edge detection methods factor in smoothing and noise reduction and are capable of being implemented quickly with OpenCV.

Edge detection algorithms with OpenCV:

The following information about edge detection algorithms is according to articles from geeksforgeeks.org and medium.com

1. Gradient detection algorithms use matrix multiplication to smooth and calculate the derivative of each pixel in the X and Y directions. The gradient magnitude is then found by the formula $\sqrt{((\frac{d}{dx})I)^2 + ((\frac{d}{dy})I)^2}$, which uses the derivatives of the X and Y directions. A threshold is then applied to ignore smaller gradient magnitudes to show only the most prevalent edges in the image. The different types of Gradient detection algorithms are primarily differentiated by their derivative filters.

    a. Sobel Operator uses two filters defined as:

$(\frac{d}{dx})I$

-1	0	1
-2	0	2
-1	0	1

$(\frac{d}{dy})I$

-1	-2	-1
0	0	0
1	2	1

    b. Roberts Cross Operator uses two filters defined as:

$(\frac{d}{dx})I$

1	0
0	-1

$(\frac{d}{dy})I$

-1	0
0	1

c. Prewitt Operator uses two filters defined as:

$(\frac{d}{dx})I$ $(\frac{d}{dy})I$

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

2. Laplacian Detection:

a. Laplacian of Gaussian (LoG) uses Gaussian Smoothing to reduce the noise of the photo using a Gaussian filter: $\frac{1}{2\pi\sigma^2}e^{\frac{x^2+y^2}{2\sigma^2}}$. It is then convolved with the Laplacian Operator, $\frac{d^2I}{dx^2} + \frac{d^2I}{dy^2}$. The second derivative is used to define areas of contrast even if the first derivative shows a negative value. The filters are approximated with:

$(\frac{d}{dx})I$ $(\frac{d}{dy})I$

0	-1	0
–1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

b. Difference of Gaussians (DoG) is found by subtracting two images of different Gaussian standard deviations (σ). It is faster to compute than Laplacian of Gaussian, but it can be less reliable.

3. Canny Edge Detection

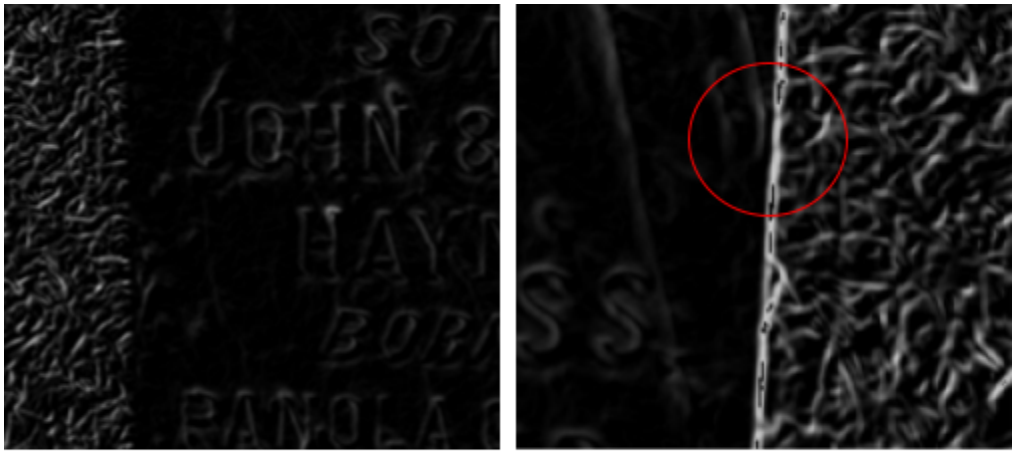
a. This is a more complex and accurate version of edge detection with five steps: Smoothing, calculating gradients, non-maximum suppression, double thresholding to reduce extraneous detail, and removing weak edges not connected to a prominent edge. It is also the edge detection method currently used in the VR project. Because of Canny's ability to remove unimportant edges, it will prove to be a more reliable edge detection method. All the headstones have text that is detectable under Sobel. Additionally, Canny had a decent amount of success defining the tombstone edges in the previous group, so changing the edge detection method is unnecessary.

Prototype of automated texturing

Before working on the main project, team 2 of the Cemetery VR project decided to create a rough prototype of the system to understand the nuances of image manipulation and computer vision. The previous project groups had issues with image stretching, incorrect cropping, and badly captured pictures. A rough prototype would be helpful to identify why these issues are occurring and ways to work around them. For example, image stretching could be an issue caused by a model's UVs being made incorrectly. Additionally, issues with the AI model are also possible.

Edge detection

To compare the accuracy of Canny to a simple edge detection algorithm like Sobel, a quick program was made. They were tested on the same images and the same Gaussian kernel size. Due to being a simpler algorithm, Sobel is unable to filter out small edges. In the figure below, Sobel has shown to display faint wisps of every point of contrast in the image. Considering the background is black, any contour detection algorithm would have difficulty identifying noteworthy edge lines. Additionally, contour detection might create outlines around edges, resulting in lines with small black centers. Due to the inability to deal with large amounts of edges, Sobel and Laplacian cannot be used.



[43] Issues with Sobel edge lines

Canny, although more complex than edge detection filters, is simpler to program. CV2's library has a Canny function that only requires the image and the threshold range. The edge lines produced are simplified to be as thin as pixels. Thresholding for Canny is also a lot more effective as it ignores the edges for letters and faint changes in color. The tombstones have slight changes in color and some indentations that create multiple points of contrast, as evident in the Sobel edge detection. A major issue faced by all the edge detectors is noise and grass. Gaussian blurring is effective at removing noise as small as pixels, but it cannot remove the noise caused by large patches of grass. Grass has many 3D forms, shadows, and highlights that create random edges. The grass edges will also intersect with tombstones, creating uneven edges that will make corner detection impossible. The problem becomes more apparent when putting the detected edges through CV2's contour detection. To get the contour of the headstone, a "findContour" function is run. The contours can then be sorted as a way to find which contour is bigger and hence more important. However, some patches of grass randomly form longer edges than the tombstone itself.

Another issue when using edge detection is background tombstones. All the tombstones have a similar shade of white and gray, which is nearly impossible to

detect without the use of a trained AI model. Image manipulation can only isolate and mask out details if they have a different color or value. If a white object intersects with the main tombstone, it effectively changes its silhouette. This results in only part of the edges being captured before they are diverted to another object.

Given the abilities and shortcomings of edge detection, more emphasis must be placed on image manipulation. Otherwise, an AI-powered edge detector would be needed, which would cause some issues. For starters, edge detection is run locally on a computer rather than on Google Colab. Any AI model would need to be installed along with the project files to be run. Installing dependencies could create issues down the line when it comes to version control. AI models are also more complex and run slower than Canny. Canny can run edge detection, but it needs more image manipulation to mask out and remove background noise.

Overall, Canny edge detection is very good at extracting points of contrast. However, after testing Canny on grayscale images, it became evident what limitations there are when trying to extract edges:

- Areas of different colors but similar grayscale values tend to merge.
- The grass has a large variety of values that will match the values of the bordering tombstone pixels.
- The grass will overlap on top of the tombstone, creating unnecessary edges.
- The shadows of the letters on the tombstone will create unnecessary edges.

- Tombstones present in the background will match the value of the main tombstone, and Canny will be unable to differentiate the two areas.

[44] Images of headstones and versions with Sobel filtering, and Canny edge detection respectfully.

Image manipulation

Image manipulation is a necessary step taken before edge detection. Steps such as Gaussian blurring are commonplace, but for the project it is not enough preparation for edge detection to be effective. Kernel sizes ranging from nine to 50 were tested, but they either removed too little noise or too much noise, making efficient edge detection impossible.

After running a prototype of edge detection, three methods were identified as ways to reduce background noise, ignore background objects, and create seamless edges that surround the photographed tombstone:

1. Thresholding

2. Color masking

3. Object masking with AI (Bounding box and Semantic instancing)

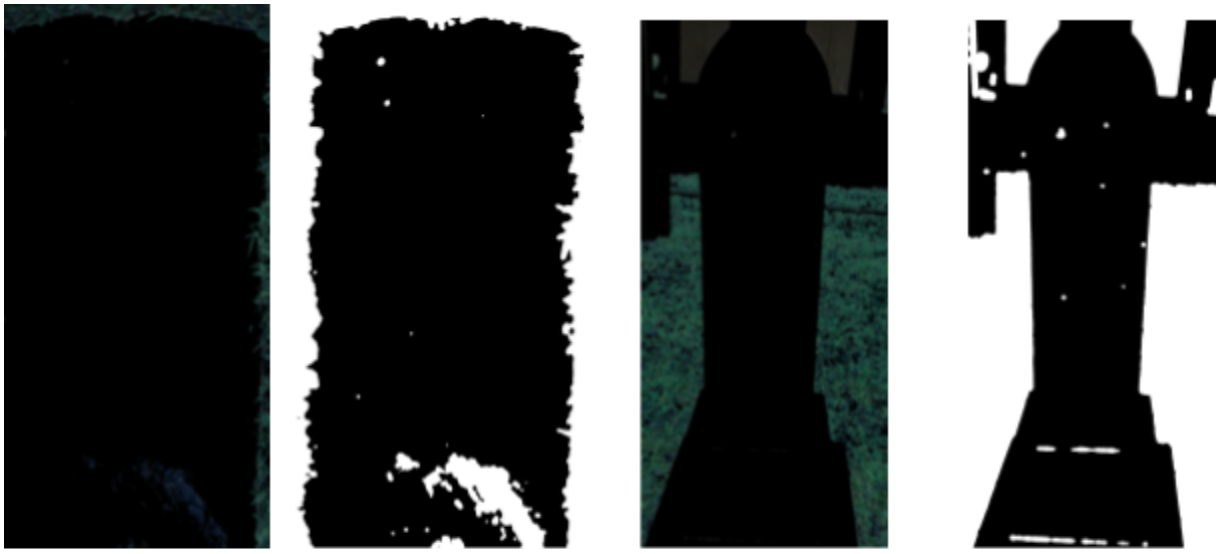
[45] Results from thresholding mask

By itself, thresholding the grayscale image was excellent at removing large blotches of noise from within the grass or from the tombstone engravings. But simple thresholding cannot tell objects of similar values apart from each other. CV2's thresholding is also meant to be run on single-channel images. For thresholding colored images, it would have to be run on each separate color channel. However, another issue occurs when an object in the background is of a similar color and value. AI thresholding is an option instead of using CV2 thresholding algorithms such as "cv2.THRESH_BINARY_INV" and "cv2.THRESH_OTSU," but it would be simpler to use information from the dectron AI model already made. Thresholding is a useful step of image preparation; however, by itself, it is ineffective.

Color masking became an important step to image preparation as a near-guaranteed way to remove grass from photos. As stated before, grass introduces a plethora of edges and information that meddle with edge detection,

contour detection, and corner detection. Thankfully, grass has an important distinguishing factor that sets it apart from white and gray tombstones, color. Most cemeteries are covered with well-tended green grass. In photo and movie editing, green is used for masking as it is hard to find pure green on people's skin or in man-made environments. Unfortunately, one cannot simply mask out all green color from a photo, The RGB/BGR color system is made to be a combination of the three color channels. Masking out any channel with green in it will mask out the tombstones as well. Colors close to gray, particularly light gray, have large quantities of green in them. Grays and whites need an even combination of all color channels to maintain a neutral tone. The only way to mask out green from the image while not masking out neutral colors is to use "cv2.inRange." With this function, one can find a color channel range that defines a visually green color and isolate those pixels. A visually green color can be defined as colors with RGB/BGR values between [100,255,100] and [0,1,0].

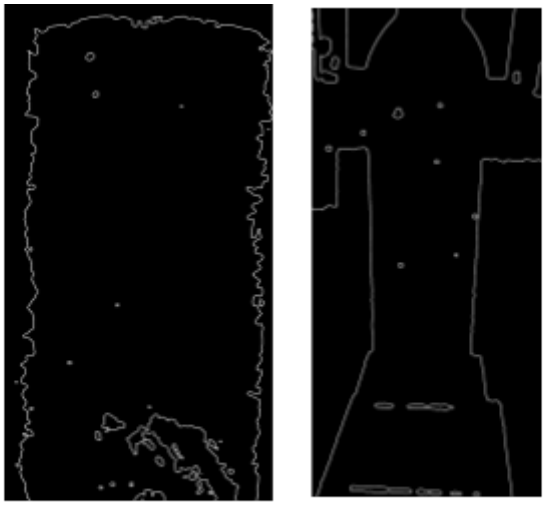
[46] RGB color range for light green to dark green color ([100,255,100] vs [0,1,0])



[47] Masking out green pixels with thresholding using to apply a cleaner mask

When color masking and thresholding are combined, they create clean edges around the tombstones in the image. It is a powerful combination for removing noise created from vegetation. Nonetheless, they still cannot distinguish when separate headstones are intersecting with each other. From testing, it can be concluded that without the use of information from a trained AI model, it is nearly impossible to use simple image manipulation to get conclusive masking for practical edge detection.

For image preparation for edge detection, two types of AI-generated information could be used: bounding box coordinates or a segmented image. Bounding box



[48] Canny edge detection with a green mask and thresholding

coordinates can come in the form of a text file. These coordinates can be used as a mask to isolate part of the image that contains the one tombstone that is captured in the photograph. As the name implies, the coordinates create a rectangle around the subject. This rectangle does not provide information on the perspective of the object. The rectangle has a height and a width defined by the maximum dimensions of the object. Even if the bounding box is not pixel accurate, it is an accurate method to remove light gray background elements.

The most accurate way to mask the tombstone from images would be through segmentation. An image of the mask would be passed to the local files for each tombstone, run through edge detection. With this method, there would be no need to retake photos that have cluttered backgrounds. If the bounding box is an insufficient mask, then the automated texturing algorithm will have to include object segmentation for more accuracy.

If an accurate edge detection method and image manipulation algorithm is made, the automated texturing system, the project will be viable for any cemetery headstone resembling a rectangular shape. This includes headstones for other cemeteries besides the St. Augustine cemetery.

Contour Detection

Contour detection uses the resulting images from edge detection. Its purpose is to find contours from the edges. An edge is simply a bunch of points where contrast exists, a contour is a line that the computer understands as a line created from connected pixels. Contour detection can be run on images, not ran on Canny, but noise would be an issue for Cv2.

The edge detection given to contour detection must contain edges that envelop the tombstone. The “cv2.findContour” function can only find edges that

already exist, it cannot connect edges that may exist. Although the “findContour” function ranks contours by hierarchy. The function returns to variables, contours, and hierarchy. Hierarchy contains the following information for each contour as an array.

1. Next: Index of the next contour at the same hierarchical level.
2. Previous: Index of the previous contour at the same hierarchical level.
3. First child: Index of the first child contour.
4. Parent: Index of the parent contour.

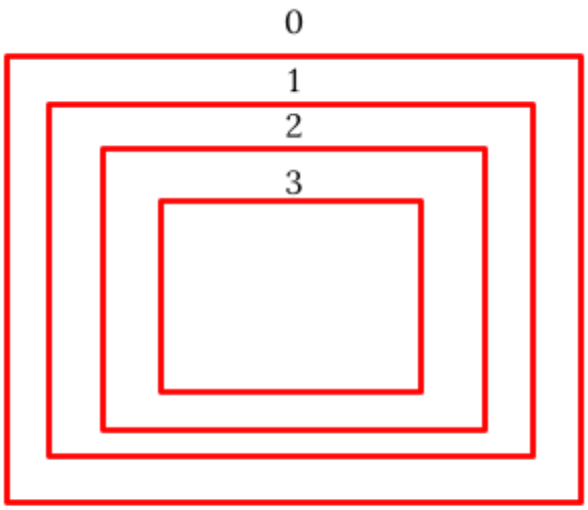
Contours are defined as a child contour if it is fully inside another contour. It is important to note the hierarchy information is defined by the retrieval mode used. The retrieval mode used for the prototype is “cv2.RETR_EXTERNAL.” It ignores contours if it is a child contour; therefore, if the correct contours are found, it should find only the contour that makes up the headstone.

Retrieval mode, “cv2.RETR_LIST,” ignores the hierarchy of found contours. Specifically, parent-child relationships do not exist. All contours would have a negative one for their third and fourth term. The contours would be ordered in terms of a by the first and second terms. As stated before these terms determine the next and previous contours in the list. The index of the contour in the two dimensional array is also the number of the contour itself. So contour zero would be the first contour in the list with an array index of zero. However, since there are no previous, parent, or child contours due to the nature of the RETR_LIST, the other values are negative ones.

Index	Next Contour	Previous Contour	Child Contour	Parent Contour
0	1	-1	-1	-1
1	2	0	-1	-1
2	3	1	-1	-1
3	-1	2	-1	-1

This retrieval mode is meant to get all contours in the scene if hierarchy does not matter for the code. However, due to the nature of hierarchy, it would be a helpful feature to remove smaller contours. For example, in the case where contours for small blotches in the tombstone are detected, they can be easily removed as they would be found to be the child of the a parent contour. In this theoretical scenario, the tombstone would be the parent contour of the smaller contour.

Retrieval mode, “cv2.RETR_CCOMP,” is a two-level hierarchy method where a contour is considered a parent if it has an contour in it. Even if the contour is a child to another contour it can be labeled as a parent anyways. The difference being, the retrieval method does allow for a child of a child to have the same parent. Additionally, a child cannot have another child labeled as a child. If there were 4 contours embedded into on another, the contour second from the outside would be a child, but the child of that second contour would be a parent of the innermost child node.



[49] Contours numbered by index for the table below

Index	Next Contour	Previous Contour	Parent Contour	Child Contour
0	1	-1	-1	-1
1	2	0	-1	-1
2	3	1	-1	-1
3	-1	2	-1	-1

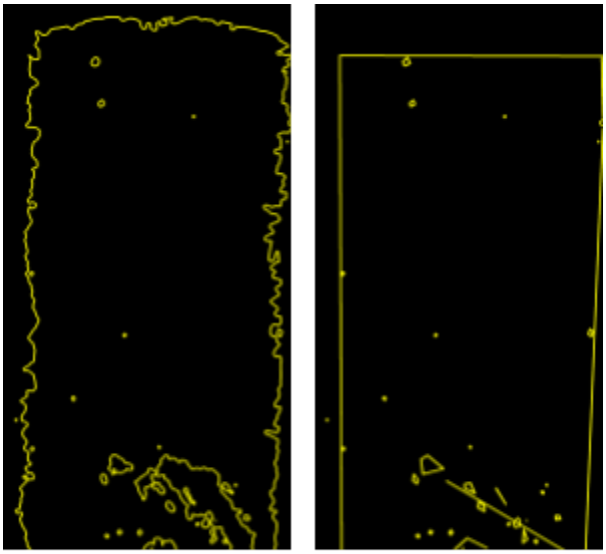
Retrieval model “cv2.RETR_TREE” creates the full line of hierarchy. Similar to a family tree, it will keep track of the child of every parent and the child of every child contour.

Contour Approximation

A parameter of the drawContour function is the approximation method. The approximation parameter determines if and how the contour should be simplified. “cv2.CHAIN_APPROX_NONE,” draws along all points of a photo’s edges. “cv.CHAIN_APPROX_SIMPLE” compresses the amount of data to only the ends of an edge’s line. So in the case of a square-shaped edge, it would only keep track of four points. Without simplification, the program will create a tracking point for every small group of pixels along the edge, turning four points into several hundred points.

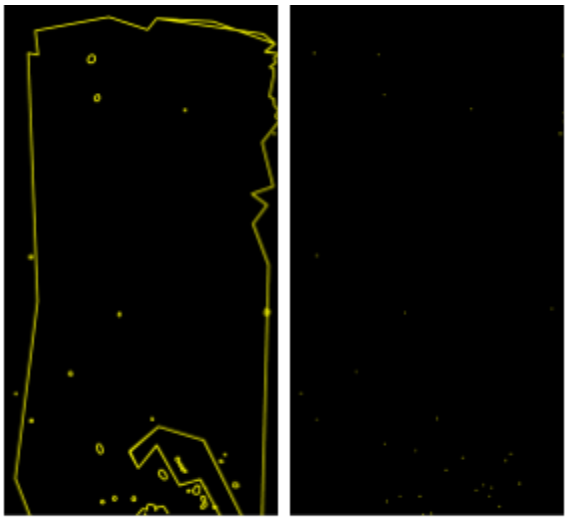
The more complex method of contour approximation involves configuring the contour approximation to the desired level of detail. This is done by using “cv2.approxPolyDP.” The contour approximation algorithm is a function that takes the parameters: the contour, the epsilon value, and a boolean value to determine whether or not the simplified contour should be closed. Epsilon’s value is determined by multiplying a small decimal number by “cv2.arcLength,” which takes the contour and a boolean value for whether the contour should be closed. As the name implies, “arcLength” takes the length or perimeter of the contour input.

The output of “cv2.approxPolyDP” when used as a parameter of “cv2.drawContour” is a simplified version of the contours already found. This is useful for removing unnecessary corners and arcs, simplifying the contours to a much smaller number of points.



[50] Contours with vs without approximation

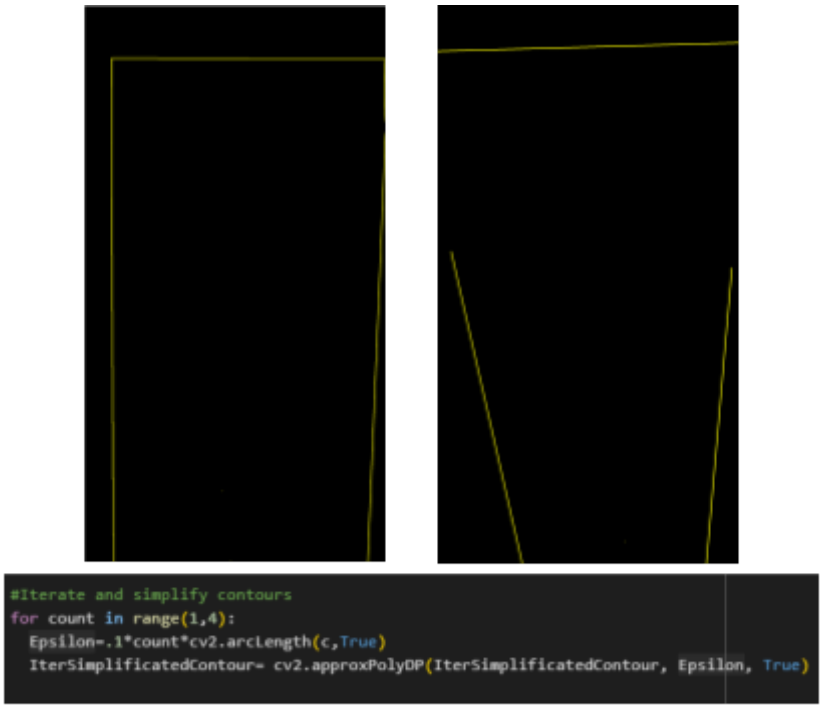
The figure above compares the two contour outputs with and without "cv2.approxPolyDP." The epsilon multiplier is the decimal number multiplied by the arc length to determine the accuracy of contour approximation. When this value is set to 0.061 is reduced by 10% while retaining the accurately straight contours. When the epsilon is set to values above 0.1, the algorithm removes lines altogether, creating dots or small circles. Large epsilon values are likely unusable given how much information is removed. Approximation with epsilons above 0.1 may prove useful for corner detection, as the dots found are the approximate end points of contours.



[51] Contours with an epsilon of 0.011 vs 0.61

After further testing, large epsilons were found to have issues with layering copies on contours on top of each other. This would make a polygon not found in edge detection or the original image. To

prevent this, a loop was made to iterate multiple contour approximations that iteratively increase over time.



```
#Iterate and simplify contours
for count in range(1,4):
  Epsilon=.1*count*cv2.arcLength(c,True)
  IterSimplificatedContour= cv2.approxPolyDP(IterSimplificatedContour, Epsilon, True)
```

[52] The results of iterating through the contours to remove small arcs and slowly increase simplification

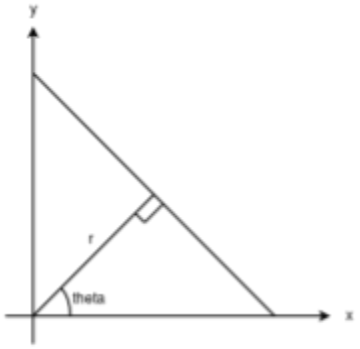
For the past groups, the automated texturing used a contour approximation to estimate the location of corners. This method could work if contour detection were capable of ignoring insignificant contours. If a continuous edge does not encircle the tombstone, then isolating those edges will prove difficult. The goal of contour detection for the future of the project is to use hierarchy to remove all child contours accurately. The accuracy of hierarchy relies on whether the contour of the tombstone is enclosed.

Additionally, there may be cases where a contour becomes disconnected because it clips into the end of their PNG file. To remedy this, padding can be added to the image before edge detection. Padding essentially increases the size of

the photos without changing or scaling the original pixels. It adds pixels to the ends of the photo by replacing it with a flat color, by tiling the original image, or by repeating the pixels at the image border. A flat black color can be added to the image after the grass is masked, so that if a corner of the tombstone peeks off the edge, it will create an edge for it. There are no plans to move away from cv2's contour detection as it has no issues. After programming the edge and contour detection, most issues likely came from the corner and edge detection. Although contour detection can be used to find the endpoints of contours and get corners, having another algorithm to find corners would be better.

Hough Line transformation

According to geekforgeeks.org, Hough transformation is a feature extraction method capable of detecting shapes that can be represented with a mathematical form. In the case of this project, Hough line transformation is a possible method to extract straight lines from a given image. A line can be represented by a linear equation $y = mx + b$, but in the parametric form, it is $r = xcos\theta + ysin\theta$. This allows cv2 to easily track the angle of Hough lines along with the length of the vector.



[53] Example graph in parametric form by geeksforgeeks.org

Before using Hough lines, the image has to be run under some sort of edge detection, like Canny. The input needs to be a black and white image, as the Hough line function cannot accept images with three color channels. Inputting a colored image will result in an error. However, to print the lines for testing, the image has to be colored. Additionally, to print lines, the lines have to be manually drawn by a for-loop because the output of Houghlines is not an image

but a list of coordinates in parametric form. “cv2.HoughLines outputs a 2d array with the values:

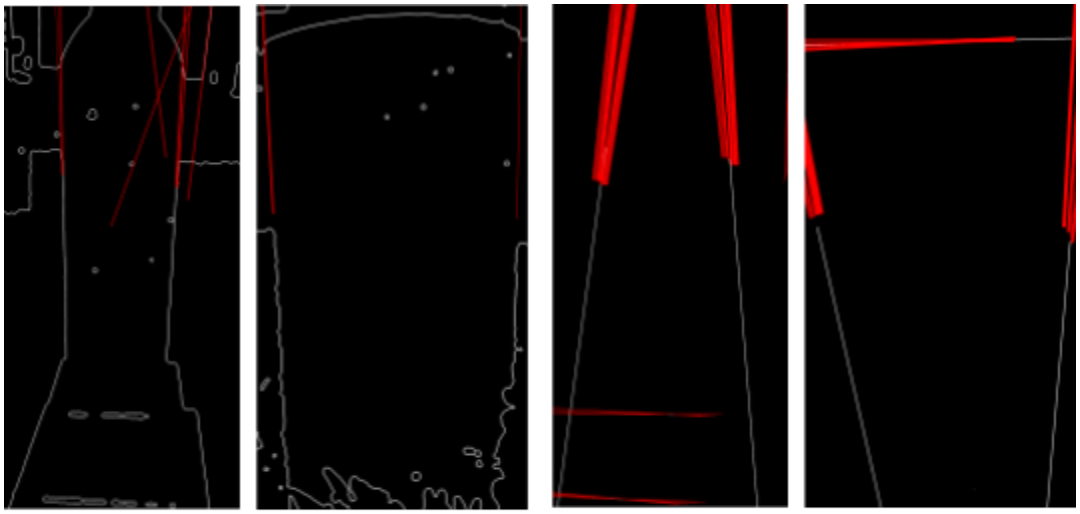
- Rho - This is the radius or length of the line drawn.
- Theta - This is the angle of the line.

To get the endpoints of the line in a Cartesian coordinate form, the rho value is multiplied by the $cos(\theta)$ to get x_0 and multiplied by $sin(\theta)$ to get y_0. Afterwards, the line is extended by adding or subtracting from the original x and y values by a larger value of $cos(\theta)$ or $sin(\theta)$ to get x_1, x_2 and y_1, y_2.

According to OpenCV’s official documentation, “cv2.HoughLines” takes the parameters for:

- A resulting image from contour detection, which must be a grayscale image from contour detection
- The pixel step is the resolution of the accumulator in pixels. This is typically set to one.
- The degree step is the resolution of the accumulator in radians. This is typically set to one degree in radians by the division $\pi \div 180$
- The threshold for the number of intersections needed to detect a line. Increasing this value reduces the number of Hough lines found.

Creating the accumulator array, also known as the parameter space, is done as the first step to the Hough Transformation. The accumulator array is a 2D array whose columns define the accuracy of the Hough transformation. If the degree step is one degree, then there would be 180 columns in the array. The pixel step defines the number of rows in this array.



[54] Results of Hough line transformation without vs with contour approximation

Hough line transformation won't be used in the prototype of automated texturing, however, it was important to consider it. The previous groups use this method to finalize the contours of the tombstone. Hough line transformation does not do well on the raw edge detection or unsimplified contours. After running on approximated contours and raw contours, it is apparent that the lines of the input image cannot have a variety of curved arcs. In the figure above, the grass at the bottom of the tombstone creates a large number of arcs, so that when the Hough line threshold was reduced, the predicted straight lines create a mess of chaotic straight lines.

Corner Detection

A corner in computer vision is defined as a region with a large amount of directional variation. A change in direction typically occurs at the point where edges intersect. To make judging the direction of edges easier, contour approximation needs to be done first, otherwise, the amount of arcs will cause the detection of nonexistent edges. Two possible corner detection methods are to use the findContours with approximation, or by using Harris Corner detection.

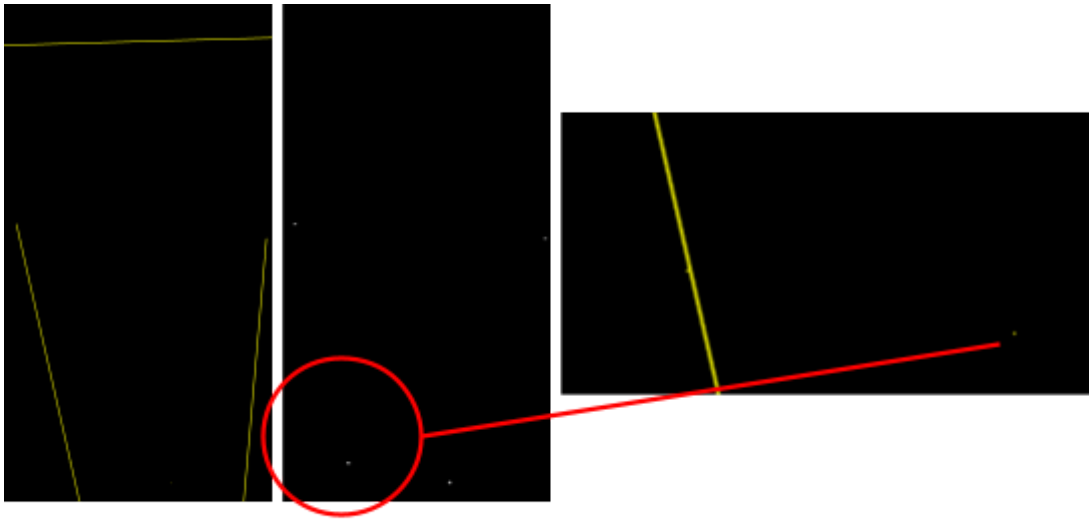
Contour Corner Detection

If the contours are reduced to a group of four arcs, the endpoints that are saved from the findContours can be used as the corners of the object. The act of contour approximation reduces the number of arcs in the contours. By default, without approximation, findContour saves a coordinate for every pixel of an arc. Approximation removes the number of data points between the ends of the arc, therefore making it possible for findContours to get the corners of an object.

Harris Corner Detection

As stated by OpenCV documentation, “cv2.cornerHarris” is a corner and edge detector that uses the “intensity of displacement in all directions.” The detector equation is defined by: $E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y]^2$. The approximation of Harris Corner detection is $E(u,v) \approx \sum_{x,y} [u\ v]\ M[\ {}^{u}_{v}]$. Because OpenCV handles the algorithm for Harris Corner detection, there are only four value parameters of concern:

- A processed grayscale image that has been through either thresholding or edge detection to reduce information that must be processed.
- Blocksize is the size of the pixel group analyzed for corner detection.
- kSize is the aperture parameter for the Sobel derivative.
- Harris detector free parameter (k) that ranges between 0.04 and 0.06.



[55] Harris corner detection (the middle image is the output) ran on simplified contours..

Harris Corner detection fails to get the endpoints of contours if the end of the contour is not visible on the image. In the figure above, the right and middle images show that the only corners identified were found because very small contour arcs were there. This provides another motivation for padding the borders of the images before edge detection to guarantee this won't be an issue for the main project.

Perspective Warp

Flattening the image, assuming four correct points have been obtained, is straightforward. Use “cv2.getPerspectiveTransform” and then apply the transformation with “cv2.warpPerspective.” These two functions handle the plethora of matrix math required to flatten an element of an image. “cv2.getPerspectiveTransform” outputs a matrix referred to as “M” that can be used to warp a photo to have a different perspective. A problem arises regarding the dimensions of the new image.

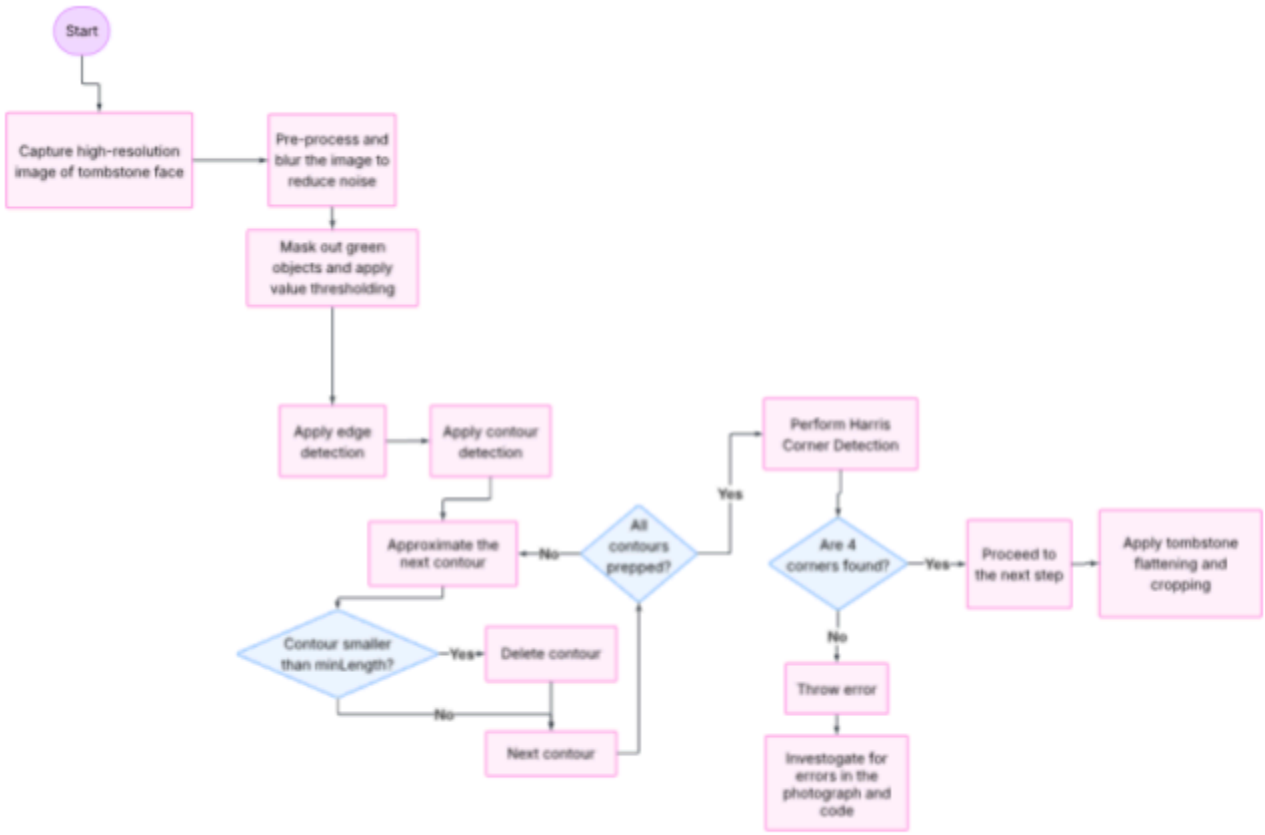
In the automated texturing project, the previous groups had all standard tombstones flattened to 1005 pixels wide by 1590 pixels tall. Tombstones that have

a smaller or larger height-to-width ratio will have issues of image warping. The best way to get the desired dimensions is to use the bounding box coordinates from Detectron or use the maximum values of the tombstone corner coordinates. The downside of this method is, images with varying camera angles will have varying output sizes.

[56] Standard headstones do not all have the same ratio of height and width.

Conclusions from the prototype

Unfortunately, the flattening function of the prototype is not yet complete, there are plans to finish it during summer break before starting on the main project. The current prototype allowed for many learning opportunities with computer vision and OpenCV, and it showed what features the main project may need to flatten images more accurately. Additional preprocessing, such as masking out grass and padding the tombstone photos with black are potential solutions to incorrect corners. The Harris Corner Detector could also be an alternative used in the project instead of using the endpoints of contours to find corners. Lastly, to avoid cropping and improper image warping, the desired dimensions of flattened tombstone photos will be made to match the aspect ratio of the tombstone itself.



[57] Breakdown of the prototype for automated texturing

Executive Summary

This section of the design document will outline the processes and tools for integrating Docker containerization and unit testing to the Cemetery VR Project. The primary goal for my initiative is to streamline development workflows for the team, ensure consistent build environments, and improve the quality of code integrated into the main branch through automated testing. Through the utilization of Docker images, we can create a remote environment for portable and reproducible builds, while unit testing tackles the reliability and stability of the actual codebase.

The following areas will be covered

1. Github Best Practices and Team Collaboration
2. Docker Image Creation
3. Containerization in Unity
4. Continuous Integration
5. Unit Testing

GitHub Workflow

- General Overview

  The goal of establishing a GitHub workflow is to ensure an environment of cohesive collaboration among all team members and their separate responsibilities. Through both personal experience and research on real-world practices, I developed guidelines to accomplish the goal of streamlined development, improved code quality, and seamless integration of changes.

- Structured Approach

  To accomplish a consistent workflow, each team member will follow the same structure for making and testing their updates. For all stories involving updates to the codebase, a custom branch will be created from the main branch. Each member will work on their own branch to ensure no clashes in updates are possible. Branch names will follow the naming

convention of their JIRA story. If the JIRA story SCRUM-43 Upload Docker Image is in development, the branch name will follow the format SCRUM-43_Upload_Docker_Image, or some similar description. The SCRUM-## will be a hard requirement for the branch name, however. The custom branch must show all unit tests passing before it can be merged in, unless a bug is found in the unit test itself. The main branch will serve as the default branch, and will never be updated directly.

- Clear PR descriptions

When updates are ready for approval, a member will create a Pull Request (PR) in GitHub for their custom branch. PR's will also follow a specific structured approach. The PR title will contain the JIRA Story as well as a few words describing the work done. The PR description will elaborate more on the work that was completed. An example of acceptable description would be below.

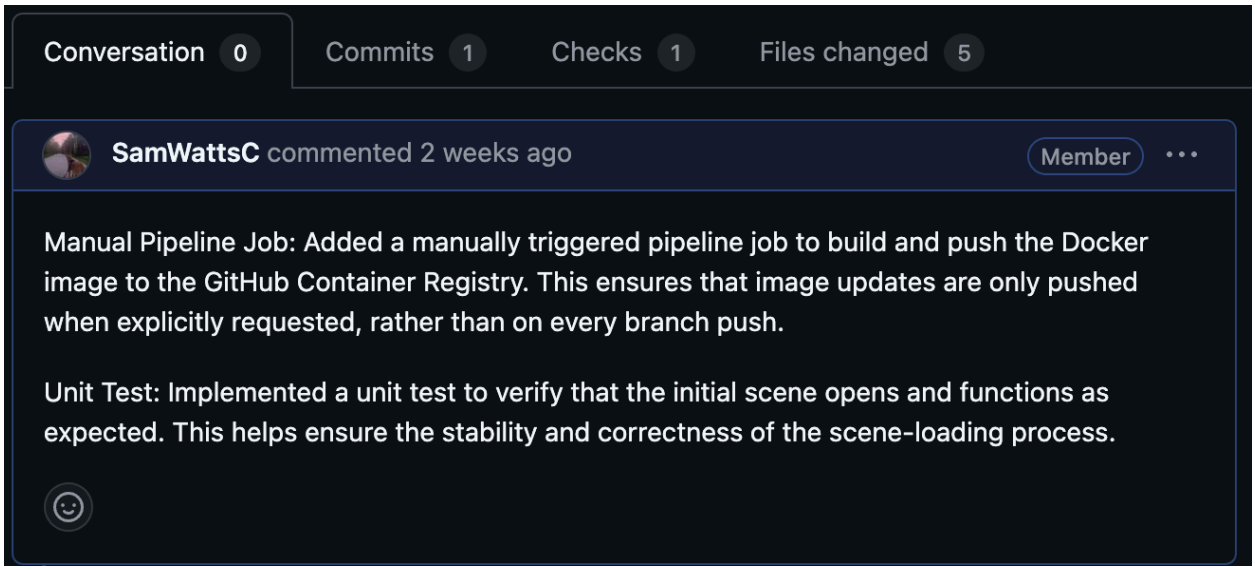


Fig.[58]

The description is concise, clear, and establishes all the updates made as part of the story, along with reasoning for the update if necessary.

- Clean Commit History

  A clean commit history plays a big part in the readability and documentation of code updates. Members should be able to find out if PR was associated with the specific update they are interested in, and going to the PR will contain a description of the updates made, links to the passing unit tests, and any extra information needed to understand the goal and reasoning behind the work done. Part of accomplishing this goal includes following a specific naming scheme for commits. They will include the JIRA Story and a few words describing the work, very similar to the PR description.

- Mandatory Rebasing

  Rebasing will be mandatory for two primary reasons, to enforce a clean commit history and to ensure PR's are compatible with updates made to main during the course of the story's completion. With software development there are usually tens to hundreds of commits before work is functionally ready for merging. Once a PR has reached functional approval, it will be rebased down to one commit. This ensures that all updates associated with a story can be linked to one. This will make it easier for future work to reference old story updates to the codebase. The other important reason, ensuring PR's are compatible with updates made to main, calls for rebasing to be done with the main branch. A member will use the command git rebase origin main while on their custom branch. This will apply your custom branch changes on top of the latest updates to main. This is useful for keeping the custom branch up to date and is consistent with the goal of maintaining a cleaner git history. Research was done to compare rebasing vs. merging and rebasing accomplishes the team's goals in a more comprehensive way.

- Peer Review Process

  At least one other team member must approve the PR before it can be merged. If two team members collaborated on a branch through pair programming, a third-party reviewer must approve the PR. The goal of this process is to reduce biased inspections and maintain code quality and consistency. This process will also ensure that other git workflows (PR description, rebasing, etc) are followed.

- Hotfix Workflows

  In the unfortunate event that there is a large bug introduced into the main branch that halts development, a hotfix workflow is in place. The hotfix branch will follow a similar naming scheme to other custom branches, but will always start with SCRUM-00 followed by _Bug_Description. This special branch will still require a PR, but with only a brief description of the bug fix. An impartial third party will approve the hotfix to be merged in, same as a normal review process. This hotfix format will ensure updates, though quick, follow standards that should prevent extra bugs from being introduced. A quick update will also prevent disrupting other ongoing stories or leaking into further sprints.

Docker

- DockerFile Design

  The DockerFile defines the environment and dependencies needed to build and run the Unity project in a container. UnityHub contains preconfigured docker images to be used by Unity projects in GitHub. As long

as we know which version the project uses (2021.3.8 in our case) , we can choose from multiple images stored on the hub. The following link contains the type of unity image, https://hub.docker.com/r/unityci/editor/. From here there are multiple tagged versions to choose from, ensuring we are able to find an image that suits the project's specific needs. The base image defines the environment and dependencies needed to run the Unity project in GitHub's pipelines for unit testing. The base image chosen: unityci/editor:ubuntu-2021.3.8f1-linux-il2cpp-3.1.0. Optimizations needed to be included to create a docker image that fit in the storage constraints that come with the free version of GitHub. Unnecessary components included PlaybackEngines for Android, IOS, TizenPlayer, and WebGLSupport. Removing these components allowed the docker images to stay within the storage constraints while not compromising functionality of the container. Workspace is set as the working directory for the container. The container also defaults to running a bash shell.

- Building the Image

  A big goal with this project is scalability and automation of work. This goal remains the same for docker images. A yaml script has been used to create a manual pipeline job, building and pushing a specified DockerFile to the remote repository as a usable package by all project members. Running as a manual job was important, as unnecessary resources would be used to rebuild the image every time a project was pushed otherwise. The reusability of this script is vital, as any future image updates by this team or another is as simple as updating the DockerFile in the root directory. Then the "Build and Push Unity Docker Image" job can be manually run to rebuild the new docker image. The docker build command is used to build the

image. This process can be seen in the GitHub pipelines through GitHub actions. The flow of the script process can be seen below.

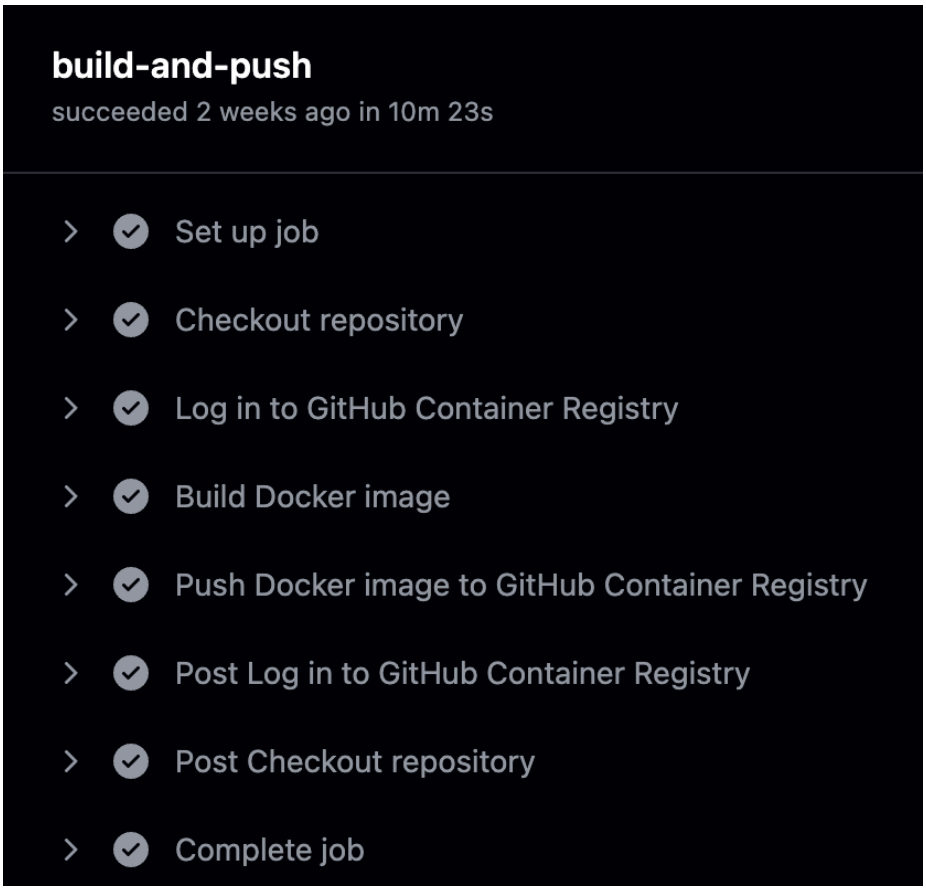


Fig.[59]

- Pushing the Image

The docker-publish.yml file currently automatically pushes the image to the GitHub repository. The build-and-push job runs using ubuntu, and has permissions to read and write to the repository. These permissions are key, as without them the job will be blocked from pushing the docker image to GitHub. Using “Secrets” in GitHub actions allows the job to access privately stored information without exposing that information to viewers of

the repository. The job uses the username github.actor and password secrets.GITHUB_TOKEN to authenticate and make the needed updates. Docker images, when pushed, have to use a name that defines what they are while also being all lowercase. The current image uses repo_name/unity-custom:2021.3.8f1-linux-il2cpp-3.1.0 as the name.

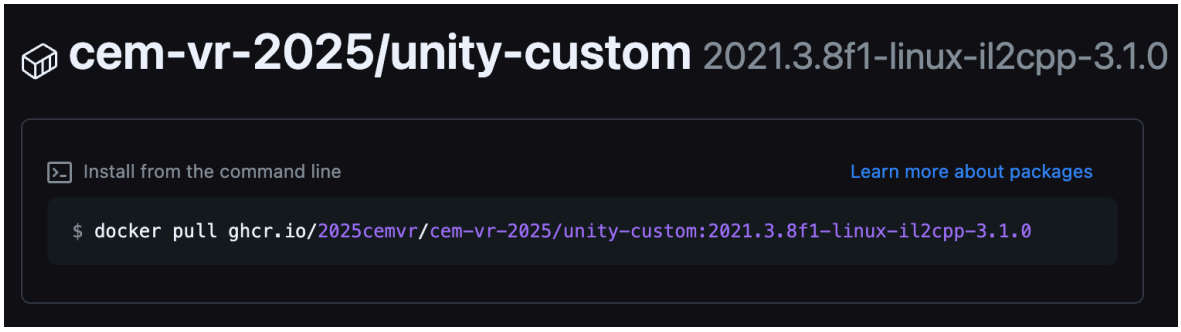


Fig.[60]

- Image Versioning

  Image versioning is a key part of keeping multiple docker images tracked and easily accessible. This feature will be implemented in the near future, as needs for the project continue to evolve.

- Best Practices

  With Docker images, there are a few key practices to keep in mind for consistent, reproducible, and safe project development. Minimizing the image size is one. These images are often multiple gigabytes of information. With storage constraints like we have with Cemetary Virtual Reality, every available byte matters. Unneeded dependencies and components should be removed from the image before pushing when using pre configured images. Another element of docker images to account for is layers. With each instruction in the DockerFile, a new layer is created. Layers get cached,

which is great for rebuilding images with some reusable components. These layers can however take up precious storage space. Related commands should be grouped together under one instruction to minimize the number of layers. This is present in our current DockerFile, as multiple commands are grouped under one run instruction. The format follows our code below.

```
# Remove unnecessary files to reduce image size
RUN apt-get update && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
```

Fig.[61]

Security should be kept in mind as well. Containers are run as the github user, not root. If a container is run as a root user and a malicious actor gains access to said container, they have greater accessibility to finding vulnerabilities and breaking out of the container. Documentation is very important with this process. A README will be maintained for this project, where a general overview of key points on the docker images will be explained. In the DockerFile and yaml scripts specifically, there are comments above each section that performs a new action. This easily breaks down the process for a clean development history. Anyone looking back at the work done can tell what would need to be updated vs. what should stay.

Containerization

- **Why**

  Containerization plays one of the most important roles in creating a consistent environment across all users for development, testing, and deployment. We want to avoid the “it worked on my machine” issue that can arise from working in separate environments. Containers also isolate the Unity project and its dependencies. There’s no risk of our project's elements conflicting with other software on a user's computer. These containers are also scalable. If needed for this project, they can be scaled horizontally to handle increased workloads. An example of this would be running multiple instances of the Unity Editor for unit testing. A big focus is on this CI/CD pipeline. Deployment is not part of the goal for this stage of the project, but continuous integration is. Containers integrate very well with CI pipelines, as they enable automated builds and tests. These will happen on every push to an existing PR.

- **Challenges**

  There are some Unity focused challenges that may arise as development continues. Containers are often slower than running the project natively. The pros of consistent development environments outweighs this specific downside. They also introduce more complexity to the development of the project. This will add extra elements to the project, increasing the “spider-web” of development. With so many interconnecting pieces, it can be harder to understand what is doing what. The solution to this challenge has already begun. Clear documentation both in the code and the README will help to break down the complexity into manageable bites.

- **Workflow**

  The workflow is intertwined with the Docker Image workflow. A DockerFile is created that specifies the image details. A script is generated to build and push the docker image, using the details in the root level DockerFile. The image is built, the container is run, the image is pushed to the GitHub registry, and now the container can be used to run the Unit Tests with GitHub actions.

Continuous Integration

- **Why**

  Continuous integration has brought many benefits to the development life cycle in recent years. This process gives developers early feedback on how their updates would impact the main branch before officially merging in. This also cuts down on review times, as developers can work out the obvious breaks without a group member's oversight. These automated builds and tests are an important prerequisite for automatic deployment as well. This is a large part of why implementing continuous integration is important for Cemetery VR. Deployment is in the future once Dr. Giroux is ready to deploy the game for individuals to use. Though this deployment is not part of the tasks for this Senior Design team, setting the stage for it is just as important.

- **Pull Requests Role**

Pull requests will activate pipelines through the continuous integration principles. For one, they can control when pipelines are run. To keep pipeline histories clean, they will only run on updates to a branch that has an existing pull request. PR's also provide a quick and easy way to compare our team's custom branch changes to the main branch. This organization will better allow our team members to determine which updates are the culprit for a failing test case run by continuous integration. There is a "Checks" tab for us to easily access the most recent tests. Another pro to our team utilizing pull requests to merge changes in is that PR's have a useful section for checking if changes are compatible with the main branch. Should another team update the main branch in a way that affects other members' working changes, the PR will reflect this conflict, and in turn prevent the merge from taking place. This part of continuous integration will prevent merge conflicts for stalling our development.

Unit Testing

- **Importance**

It is easy to write "bad" unit tests. The real value and importance comes from creating useful tests. Given the occurrences of components breaking like the query tool to access the Cem VR database, or the lack of optimizations (trees with 20x the polygons needed) the importance of code consistency and quality will be upheld by these important unit tests. There is a large focus on maintaining functionality and optimizing current processes. Unit testing provides early defect detection, improved documentation, and ensures code quality remains consistent and maintainable.

- **Unity Licensing**

Unity licensing was initially looked into for the purpose of running unit tests. Documentation such as https://github.com/marketplace/actions/unity-builder led me to believe that personal licensing may be necessary to automatically kick off successful unity unit tests. GameCI, an open-source set of tools and workflows, has been designed to make the unit testing process for Unity projects easier to integrate. Unity currently does not support an in-house CI/CD solution, and GameCI filled in the hole left by this crucial missing piece of software development. They provide the steps for setting up the licensing, necessary secrets, files, etc. This information is stored here: https://game.ci/docs/github/activation/. GameCI also contains Docker images with pre-installed unity. We already have an installed Unity image in Github, but using a GamCI image in the future may be more ideal in the future. It has a special, potentially useful feature in allowing developers to automatically build their Unity projects with different operating systems. These are included but not limited to Windows, maxOS, and Linux. It is important to note that as of v3.4.0, not all build types are fully functioning, and code coverage is relatively weak. See below for more details.

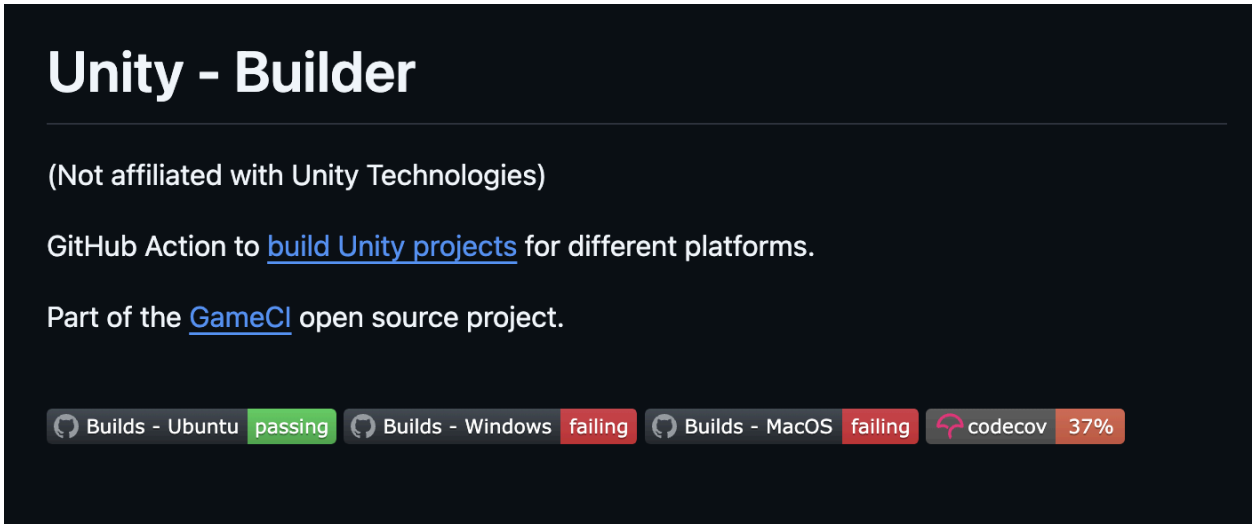


Fig.[62]

- **Testing Frameworks**

  Testing frameworks consist of two major players. These are the Unity Testing Framework (UTF) and NUnit. The UTF is a great all-around framework. This is built-in to Unity and is the recommended way to write unit tests. It can run in both edit and play modes for testing the Cemetery VR project. The edit mode is useful for testing standalone scripts, logic, or data. Essentially, testing edit mode will be great for validating functionality of project elements that do not require the Unity engine to be running. There is also the play mode. This mode allows github actions to run tests with the Unity engine. A simulated game environment is started up. From inside this simulated environment, a series of gameplay mechanics can be tested out. These tests include gameplay mechanics, collisions, interactions, etc.

- **Writing Tests**

  Writing the tests is done in the newly created .github/workflows directory. This directory has a special purpose in github. Github actions

scans this directory for YAML files and runs these files based on workflow triggers. These triggers will largely be automatic on pull requests or manually enabled, based on whether the user wants to run the test or not. Manual jobs are a great resource for automating image building, and deploying a build in the future. Each YAML file has a name. This name will show up in github actions for the project, and each name is definitive to properly track the purpose of each file. The workflow is then established. Manual jobs will follow the format below.

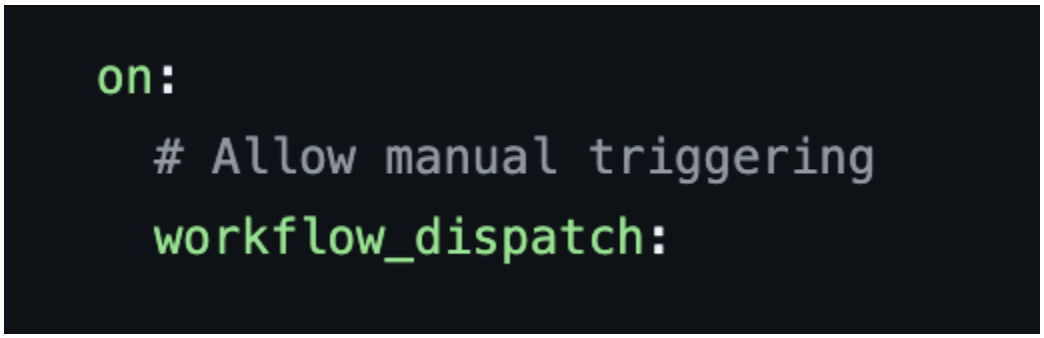


Fig.[63]

The automatic jobs run on: [pull request]. On: [push] was initially used in combination with pull requests, but this kicked off unnecessary pipelines on PR's without a simple solution for running the jobs once on a push to a branch with a PR. From here, a job name is decided. To determine which operating system to run the job on, I use the run-on command. For the Cemetary VR jobs, they are all running on ubuntu-latest, and will continue to do so throughout the course of this project for unification and simplicity. Steps are then laid out, and these steps will show up in separate sections under the yaml file results. Using a format with steps allows for easy visuals, and quickly analysis of broken sections of code. The single step will fail and that step will display an error message specific to what went wrong. The repository is then checked out. This will pull the latest code from the

repository into the runner that GitHub actions is using. Each job will use a Unity Docker image that has been uploaded to the GitHub repository. This image is locked behind login info to the GitHub Container Registry (GHCR). GitHub has the built-in github.actor and secrets.GITHUB_TOKEN to gain access to the GHCR. Now our jobs have the ability to pull our private docker image. To actually pull the image, some aliasing has to be done to ensure the job works for future forks of the project. Docker images are named in lowercase so that conversion has to be done in grabbing the repository name. As you can see, that conversion happens in the current unit testing code below.

```
- name: Pull Docker image
  run: |
    REPO_NAME=$(echo "${{ github.repository }}" | tr '[:upper:]' '[:lower:]')
    docker pull ghcr.io/$REPO_NAME/unity-custom:2021.3.5f1-linux-il2cpp-3.0
```

Fig. [64]

Using an image also speeds up the building process, as the Unity dependencies have already been built. From here, Unity is run inside the Docker image. A Unity simulation has begun. This is where the actual testing takes place. A mode is selected, again these options are playmode or editor mode. Test results are stored to .log and/or .txt files. Doing so will allow reviewers to understand testing output that may create unnecessary overhead in the event these tests are output as a script in the job itself. If the unit test produces undesired results, it is always directed to generate error code exit 1. This will make sure any errors produced create a job failure, regardless of whether running the test itself was a success. The final element of writing the unit tests for Cemetery VR is the artifact generation. Output .log or .txt files are uploaded as job artifacts. This makes them

accessible to anyone testing their code, as well as available to anyone who may need to reference an old output from the branch's history. I wanted this functionality specifically because it may be helpful in future development to want to go back and look at a test output from weeks or months ago on the main branch to investigate potential bugs, code impacts on old behavior, etc.

- **Running Tests**

  Running the tests is a largely hands-off procedure. Most are run automatically on Pull Requests. This means to see unit tests, simply pushing your code to a branch will not activate anything. The decision to use PR's as the automatic activation over pushes is a result of when the unit tests would be useful. Pushing changes can be done before a PR is created to save work and make sure it is visible. A PR draft can be created if intermediate testing is needed before officially submitting PR for approval. The other form of running the tests is with a manual workflow. To run these tests, a member will navigate to the "Actions" tab in GitHub. They will then find the name of the workflow specific to the manual job they want to run. The team members will click on this workflow, and on the right side of the page is a drop down menu for "Run Workflow". From here, members can select the custom branch they want to run the workflow for, and then click "Run workflow" to start the manual job. It is important to note that as far as I can tell, manual jobs can only be run once they are in the default branch of a project. Manual job additions will be limited as not being able to test them in custom branches is problematic for development. A workaround that I have been using is to create them as a job that kicks in on PR's, and then once functionality is tested and approved, change the activation to manual.

- **Understanding Tests**

Each job has “sub-scripts” that break down the process into easily readable and understandable error messages. Our team can take a look at the breakdown to see which specific script resulted in a failure. A targeted error message with an exit 1 code will be generated. This messaging and error system is easily accessible and noticeable for all contributors.

Fig.[65]

The error messages will outline which file and line number resulted in an error. The error message will contain the specific details as to why the error happened. This easy-to-read-easy-to-understand format for testing results will ensure that the team can quickly understand why an error is happening, where it is happening, and allow them to expedite a solution.

- **Types of Tests**

Tests will cover a variety of topics. Their focus is derived from two distinct general categories. There are simulation based tests and numerical based tests. The simulation based tests will focus on categories such as collision detection, player movement, game boundaries, etc. There are also unit tests focused on numerical/logical consistency and optimization. These

tests will focus on elements like draw count, headstone directionality, number of polygons for each object type, etc. The full lists are below

Simulation Unit Tests

- ❖ Opening scene loads in correctly

Purpose

To ensure a smooth and immersive VR experience, the application must launch directly into the correct scene, an opening scene. This helps prevent user experience issues such as a black screen, crashing, or loading the wrong scene. Establishing this behavior also serves as a baseline test for reliable scene management and overall project stability. It is crucial to validate that the scene at Build Index 0 is present, correctly configured, and works fully. By doing so, the application consistently loads the intended starting environment, supporting the project's VR goals.

Implementation

To implement this test, a PlayMode unit test will be created using Unity's built-in Test Framework. The test will begin by dynamically retrieving the name of the scene located at Build Index 0. This is defined in Unity's build settings. It will then trigger the scene loading process using SceneManager.LoadScene(0), and yield a few frames to give Unity's scene loading system enough time to finish. Once loading is complete, the test will compare the currently active scene's name with the expected opening scene name. These tests will confirm that the correct scene is both active and successfully loaded. Importantly, this test is designed to run independently and won't depend on any persistent data or objects, ensuring reliable and repeatable results.

- Collision detection on headstones to make sure object cannot walk or fall through

Purpose

This test is designed to ensure that headstone objects function as solid physical barriers within the virtual environment. It helps prevent the player character or other game objects from unintentionally walking or falling through headstones, which would break immersion. By checking for missing or misaligned colliders on these static environmental models, the test maintains a consistent sense of physical presence in the cemetery setting. Additionally, preserving proper object boundaries supports the project's commitment to historical accuracy in interactive spaces.

Implementation

To carry out this test, a PlayMode unit test will be developed to simulate physical interactions between a Rigidbody-driven test object (will be an actual player and an object like flowers) and various headstone prefabs found within the cemetery scene. The test will either load a representative headstone into a controlled test scene or retrieve one directly from an existing environment. The test object will then be moved toward the headstone from both horizontal and vertical angles to mimic walking into or falling onto the object. Using Rigidbody physics, the simulation will determine whether the headstone's collider successfully prevents the object from passing through. The test will verify that the object either stops at the collider's surface or reacts appropriately, such as bouncing off, depending on the physics settings in place. These tests will confirm that the test object cannot penetrate the collider by checking its final position or using collision detection flags. To ensure broad coverage, the test will be run

against multiple headstone variations to catch any inconsistencies in collider setup across different asset types.

- ❖ Collision detection on trees

Purpose

This test aims to verify that tree objects within the VR environment properly respond to collisions with the player or other objects. Ensuring that trees act as solid, impassable elements is essential for maintaining physical realism and a sense of immersion. Players should never be able to walk through them. During development, the test also helps catch broken or missing Collider components on these environmental assets. By preventing unintended traversal through solid objects like trees, the test avoids potential gameplay issues and contributes to the overall integrity of the scene by validating physical boundaries and constraints.

Implementation

To implement this test, a PlayMode test will be set up to simulate a physical collision between a player and a tree within the scene. A tree sample from the environment will be instantiated into a controlled test setting. A Rigidbody-equipped player will then be directed toward the tree using direct position change to mimic natural movement. Unity's physics engine will be given multiple frames to process the collision accurately. The test will then verify that the object cannot pass through the tree's collider. These tests will confirm that a collision occurred by checking the player's final position and velocity to ensure appropriate physical response.

- ❖ Collision detection on cemetery walls

Purpose

This test is designed to ensure that players and objects are unable to leave the playable area by walking or falling through the perimeter walls. Maintaining solid boundaries is essential for preserving the spatial limits of the VR environment and preventing breaks in immersion. The test helps validate the physical integrity of cemetery wall assets across different scenes and prefabs, ensuring that colliders are present and correctly configured. By catching issues like missing or misaligned colliders, it helps prevent bugs related to unintended traversal. Ultimately, it supports a realistic sense of enclosure that aligns with how the actual St. Augustine cemetery layout naturally guides people and their travel throughout.

Implementation

To implement this test, a PlayMode unit test will be created to simulate a Rigidbody-equipped object attempting to pass through cemetery walls. A wall object will be accessed directly from the main cemetery environment. The player will approach the wall from multiple angles and directions to mimic realistic movement scenarios. It will be driven into the wall using controlled movement inputs at various speeds. Unity's physics engine will handle the interactions over several frames to allow accurate collision processing. These tests will then confirm that the object is either stopped upon contact, rather than phasing through the wall mesh, which would be unrealistic. Additional checks may be included to verify that wall objects have properly configured colliders. This makes sure they're not missing, improperly typed, or incorrectly set as triggers. To ensure consistency, the test will run across multiple wall segments to validate uniform behavior throughout the environment.

- Player can move forward/backwards/sideways as directed

Purpose

This test is intended to verify that core player movement in the VR experience functions reliably in all potential directions. It ensures that input handling and physics responses are consistent for forward, backward, left, and right movement, providing a smooth and natural navigation experience. The test confirms that user input is correctly mapped to corresponding physical movement within the Unity scene. By doing so, it helps catch player control bugs such as frozen inputs, misaligned motion axes, or uneven movement speeds. Reliable movement is key to maintaining immersion within the cemetery environment, and this test also supports accessibility by ensuring consistent behavior across different input methods including keyboard and VR input devices.

Implementation

To implement this test, a PlayMode unit test will be written to simulate player movement inputs programmatically. The player controller will be instantiated in an isolated test scene to provide a controlled environment. The test will simulate directional inputs: forward, backward, left, and right, to mimic standard player movement. After each input is applied, the player's new position will be recorded and compared to its previous position. These tests will verify that the player has moved in the correct direction, that the movement's magnitude aligns with the expected speed, and that all directions result in consistent and smooth motion. This approach helps ensure reliable movement behavior across all axes.

Numerical Unit Tests

- Make sure draw count is equal to or less than 400 batches per second

Purpose

This test is designed to enforce graphical performance standards by ensuring that the number of draw calls per second remains at or below 400. Maintaining this limit is necessary for delivering smooth frame rates in VR. Even slight rendering issues will cause stuttering that ruin the immersive VR experience. The test helps catch issues early by detecting assets or shaders that introduce excessive rendering overhead. It also guards against performance drops caused by unbatched objects, overly complex materials, or poorly optimized meshes. By automating the enforcement of a rendering budget, the test supports both a stable user experience and long term scalability of the project.

Implementation

To implement this test, a PlayMode unit test will be created to measure rendering performance using Unity's UnityEditor.UnityStats or a similar profiling API. The test will load a typical game scene to mirror the conditions during regular gameplay. At runtime, it will capture the current frame's batch count from Unity's rendering statistics, using an editor-only API. The frame draw call count will then be validated against the upper limit of 400 batches per second. If the draw call count exceeds this threshold during stable gameplay frames, the code will throw an error. To avoid false positives from minor fluctuations, the test may average the draw call counts over several seconds, ensuring more reliable results.

- Headstones facing the correct direction

Purpose

This test ensures that all headstones are consistently oriented to face the intended direction, preserving the authenticity of the cemetery layout. Proper alignment is crucial to maintain realism and respect for burial practices, where headstones are typically placed in a specific orientation. By catching issues where

headstones are rotated incorrectly, the test prevents immersion breaking visual discrepancies. It also helps detect and prevent errors such as incorrect prefab usage, manual placement mistakes, and unintended rotations during scene editing, ensuring a polished and realistic environment.

Implementation

To implement this test, a PlayMode unit test will be created to iterate over all headstone objects in the primary scene(s). The test will retrieve each headstone's local rotation and compare it to the expected directional value. A configurable tolerance, such as 1 degree, will be applied to account for minor floating-point imprecision. To identify the relevant objects, the test will only focus on the direction of objects sourced from headstone-specific properties. These tests will verify that each headstone's forward vector (transform.forward) aligns within a narrow margin to the scene's intended global direction vector. For debugging purposes, logs will list any incorrectly rotated headstones, including their names and rotation values.

- Query tool can access and modify the database

Purpose

This test ensures that the query tool successfully connects to the underlying database used for headstone data specifics. It verifies that read operations, such as queries, return valid and expected data, while also validating that write operations. These include adding or editing entries, correctly modifying data into the database. By preventing runtime failures caused by connection issues, unhandled exceptions, or schema mismatches, the test helps maintain stable functionality. It also detects and prevents silent failures during insert, update, or delete operations, ensuring data integrity. Ultimately, this test guarantees that users can

reliably search for headstones, view information, and make authorized edits when permitted, supporting a smooth and functional user experience.

Implementation

To implement this test, an EditMode unit test will be created to simulate both read and write access using the query tool's API or interface methods. The test will first attempt to connect to the database. It will then perform a sample read query to retrieve existing data for a known entry, validating that the returned data matches the expected values. Next, the test will execute a write query to insert a record, followed by re-querying the same record to ensure the write operation was successful and the data was inserted correctly. Finally, the test will delete the test entry to clean up after execution. These tests will be used to check that the query execution was successful (no exceptions), that the read values match expectations, and that the write operations persist data correctly.

- Verify acceptable polygon count for trees

Purpose

This test makes sure that all tree assets used in the cemetery environment maintain an optimized polygon count, making them suitable for VR performance. By preventing the use of overly detailed or unoptimized tree meshes, it helps support smooth frame rates and reduces hardware load, especially when multiple trees are visible at once. The test establishes a quality standard for imported models, guiding artists and developers to adhere to VR performance constraints. It also automatically catches accidental addition of excessive polygon count assets, protecting the user experience from visual stutter or excessive draw calls caused by complex trees.

Implementation

To implement this test, an EditMode unit test will be written to inspect the mesh data of all tree prefabs or instances that are tagged or labeled as "Tree" or placed under a dedicated scene hierarchy. Each tree prefab or scene instance will be loaded, and the test will access the MeshFilter or SkinnedMeshRenderer component to retrieve the attached mesh. The total polygon count will be determined by checking the mesh.triangles.Length / 3. This value will then be compared against a defined maximum threshold, such as 100,000 polygons per tree. If any tree exceeds this limit, the test will fail and log the offending asset along with its polygon count and path for easy identification and resolution.

- ❖ Verify acceptable polygon count for statues

Purpose

This test ensures that all statue models maintain a polygon count that aligns with VR performance constraints, helping prevent performance drops caused by overly detailed static meshes, especially those near the player. It detects and blocks unoptimized imported assets that may have excessive polygon counts, while also enforcing consistency in asset quality across all decorative elements, such as monuments. By maintaining optimized polygon counts, the test reduces the risk of memory overload, FPS drops, or lag spikes, particularly in scenes with multiple statues rendered simultaneously, ensuring a smooth and immersive VR experience for the player.

Implementation

To implement this test, an EditMode unit test will be created to scan all assets located in the Signs Flag Pyramids directory. The test will load the mesh using either MeshFilter.sharedMesh or SkinnedMeshRenderer.sharedMesh. It will then read the triangle count using mesh.triangles.Length / 3 to determine the total polygon count. The polygon count will be compared against a predefined

maximum threshold, and the test will confirm that no statue parts exceed the acceptable polygon budget. If an asset fails, the test will log the asset's name, path, the detected triangle count, and the recommended maximum triangle count, providing detailed information for troubleshooting and optimization.

- Verify acceptable polygon count for walls

Purpose

This test ensures that cemetery wall models are optimized for VR by keeping polygon counts within performance thresholds. It helps avoid unnecessary detail on large, flat structures, which don't benefit much from high resolution geometry. By preventing unoptimized wall segments, the test reduces draw calls and memory usage, promoting seamless performance, especially in scenes where multiple wall sections may be loaded simultaneously. Additionally, it establishes a baseline quality metric for any future walls added to the environment by future teams, ensuring that all assets maintain consistent performance standards.

Implementation

To implement this test, an EditMode unit test will be written to scan all assets located in the Walls subdirectory of Assets. Each wall instance will be loaded in isolation, and its MeshFilter.sharedMesh or SkinnedMeshRenderer.sharedMesh will be accessed to retrieve the mesh. The polygon count will be calculated by dividing mesh.triangles.Length by 3. The test will then confirm that the polygon count falls below a predefined maximum threshold, with a warning threshold implemented to notify before a hard failure occurs. Any assets that exceed the threshold will be flagged, and detailed logs will be generated, including the asset path, polygon count, and the recommended polygon budget for optimization.

- Verify acceptable polygon count for hedges

Purpose

This test ensures that all hedge models used in the cemetery environment maintain an optimized polygon count suitable for VR performance. By preventing overly detailed or unoptimized hedge meshes, the test helps reduce unnecessary hardware load and avoid performance issues. It also detects and blocks any high polygon assets that may have been imported from external sources or asset packs. The test establishes polygon count standards for foliage and small plant life, supporting frame rate stability, especially when large numbers of hedges are visible within the player's field of view. Ultimately, this helps ensure that the VR experience remains smooth, even in areas with a large amount of hedges.

Implementation

To implement this test, an EditMode unit test will be written to inspect all assets within the Hedges subdirectory of Models. The test will load each hedge object and access its MeshFilter.sharedMesh or SkinnedMeshRenderer.sharedMesh. The polygon count will be retrieved by dividing mesh.triangles.Length by 3. The test will then confirm that the polygon count is below a predefined maximum threshold. If any hedge asset exceeds this threshold, it will be flagged for optimization, ensuring that all assets adhere to the performance standards set for the environment.

These tests provide a comprehensive approach to ensuring code integrity, functionality, and optimization are met.

Conclusion

For the conclusions, they have been broken up into the individual conclusions each of the team members have reached in their sections. This will detail the status of their research going into the second semester of this project,

and a brief overview summary of what they have gathered from this project thus far.

Aidan Brightman

Research conducted for this project and paper has brought attention to capabilities within the Unity Engine that had been unknown to me before this. The current status of the research this semester ends with a developed understanding of tools to optimize scenes, access databases from within the engine, write editor scripts to aid in the development process in the Unity editor, and a greater understanding of administrative tools and agile development strategies.

In addition to this, strides have been made to ensure the development we have created is done in a way that is easily replicable for future groups. Many solutions to problems could have been solved in very straightforward manners, whether it was through brute forcing the decimation of models or individually checking the headstones that were turned, however it was important to guide the team towards automating these problems so future groups will not have to sink time into those issues. As this project is going to be used for future cemetery projects, automating as many tasks as possible and creating reusable code is a priority that will be given attention to going forward.

Chad Greenspan

By automating polygon reduction while preserving visual fidelity, the tool effectively addresses the dual challenges of maintaining aesthetic quality and achieving the demanding performance standards required for an untethered–yet immersive–VR experience. The testing phase, conducted under controlled conditions, validated the tool's ability to process a wide range of asset complexities while selectively applying decimation based on user-defined thresholds. The conditional approach not only minimizes unnecessary degradation but also

streamlines the asset preparation workflow, which reinforces the broader objective of the project of scalability and future maintainability.

Looking ahead, the Batch Decimator's modular design offers a foundation for future enhancements of COTS (Commercial Off the Shelf) models. This also includes the integration of automated LOD generation and exploration of procedural decimation workflows via Blender's existing and evolving toolsets. While manual quality control will remain necessary for all models, the tool substantially reduces the burden of manual optimization across the asset pipeline. Its deployment will enable scalable and rapid asset preparation as the cemetery environment evolves with new models, ensuring that ever-evolving performance targets are met. In combination with prefab management, material consolidation, and scene optimization, the Batch Decimator will serve as a cornerstone technology supporting the project's long-term sustainability and success within the Unity VR ecosystem.

Garth Simpson

After researching the computer vision and image manipulation side of automated texturing, I feel better prepared for the coming semester. Most of my research was spent trying to find potential ways to improve the project's accuracy. The project has a solid foundation for automated texturing but it needs a few tweaks in order to remove issues such as wrapping and cropping of textures. I was also able to explore the limits of OpenCV's library to see where AI implementation might be better. For example, without AI, it is nearly impossible to isolate the main tombstone if there are other tombstones or white objects in the background. Currently, the prototype does not have transformation warping abilities, however, it will be tested before the summer semester as the AI and OpenCV sections of the prototype are combined.

Jacob Peach

By investigating the AI implementation at every stage of its operation, I not only learned where they were working, but what to look for in failure, and how that can be fixed. Through this research, I've learned a great deal about Detectron2 and EasyOCR implementation, how to spot potential issues, and how to use their outputs effectively. The problems plaguing this texture tool are nebulous, and require a fine-toothed comb to sift through all the things that could be wrong, when looking for what's actually wrong. This research is valuable to that end; by learning about the type of errors that can happen with AI object detection, I've been able to better relate them to the situation at hand, and it has assisted me in constructing my plan for moving forward.

This plan has a dual benefit, as this prototype will allow us to not only fix the issues with warping, but construct something that is easier for the end user to engage with, and with more reliability.

Samuel Watts

Through my research on how to tackle unit testing both in Github actions and through Unity's UI, I feel comfortable tackling the integration to provide our team with a consistent work environment, where any story updates will be automatically tested for quality to confirm other functions of the VR program are not negatively impacted. The GitHub Docker image setup has already been completed, following Unity's documentation, further reinforcing my confidence in the following unit testing steps. I look forward to continuing to grow my Continuous Integration skills through the following summer.

Administration

Product Backlog

- **User Story**: As a Developer, I want to be able to automate the textures for the common headstone.
  - **Task**: Research the methods already in place for automating.
  - **Task**: Research necessary tools for objective.
  - **Task**: Create a prototype for testing simple cases.
    - **Subtask**: Properly capture contour of a headstone.
    - **Subtask**: Properly align bounding boxes.
    - **Subtask**: Properly transform images with minimal visual warping.
    - **Subtask**: Properly create UV out of the resulting image.
  - **Task**: Develop the prototype to work with a larger set of cases.
    - **Subtask**: Implement color correction.
    - **Subtask**: Implement direction correction.
- **User Story**: As a User, I want the simulation to run smoothly on the untethered Quest 2 Pro while I am using it.
  - **Task**: Research current optimization issues within the Unity project.
    - **Subtask**: Create a way to calculate how many triangles are rendered on screen.
    - **Subtask**: Investigate models used within the Unity environment.
  - **Task**: Create ways to reduce triangle count in the Unity environment.
    - **Subtask**: Create automated ways to batch decimate models within Blender.
    - **Subtask**: Find replacement models for high poly-count models within the Unity environment.

        - **Subtask**: Replace models with decimated / lower poly replacements within the CemeteryAssetsFull scene.
- **User Story**: As a User, I want to be able to query the database via the menu in the scene.
    - **Task**: Research issues regarding query system breaking.
        - **Subtask**: Successfully got an Oculus Quest 2 working with Unity to test the environment.
        - **Subtask**: Go through use cases and investigate errors during runtime when interacting with the query menu.
        - **Subtask**: Investigate previous versions of the project, where the query tool was working properly.
    - **Task**: Fix issues with the query menu.
- **User Story**: As a Developer, I want unit testing implemented for the Unity project to keep a standard of care.
    - **Task**: Configure the github environment to work with continuous deployments.
    - **Task**: Integrate docker accessibilities for the project repository.
    - **Task**: Identify needs for unit testing with this project.
    - **Task**: Research existing Unity unit testing methods.
    - **Task**: Implement unit tests for the project.
- **User Story**: As a User, I want the scenery to look pretty and accurate to the real life environment.
    - **Task**: Locate and rotate headstones to be facing the correct direction.
        - **Subtask**: Create editor script to identify affected headstones.
    - **Task**: Gather models and designs needed for implementation.
    - **Task**: Research tools and methods to implement into the Unity scene.
    - **Task**: Familiarize with blender application for modeling additional unique headstones.

- ➢ **Task**: Familiarize with texturing in Unity to texture appropriate constructs.

Acknowledgements

- ❖ **Sponsors**
  - ➢ Dr. Amy Giroux
    - Provides insight into the codebase to alleviate technical debt
    - Provides imagery and reference data needed to complete the tasking
    - Guides the project direction and needed objective criteria
  - ➢ Dr. Brook Miller
    - Acted as a bridge for communication with Dr. Giroux during her month-long vacation
    - Extended help with artificial intelligence
- ❖ **Equipment**
  - ➢ Oculus Quest 2 Pro
    - Machine used to run the simulation for development and testing purposes

References

1. Unity Technologies. (n.d.). *Optimizing CAD Data for Real-Time 3D Visualization*. Retrieved March 21, 2025, from https://create.unity.com/optimizing-cad-data-for-real-time-3d-visualization

2. Giannini, F., Lupinetti, K., Monti, M., Mantelli, L., Ferrando, M., Traverso, A., Anastasi, S., Giuseppe, A., & Monica, L. (2024). A Real-time Dynamic Model in VR Environments: A Case Study. *Computer-Aided Design & Applications*, 21(6), 1076–1088. Retrieved March 26, 2025, from https://cad-journal.net/files/vol_21/CAD_21(6)_2024_1076-1088.pdf

3. GeeksforGeeks. (2024a, July 8). Comprehensive guide to edge detection algorithms. https://www.geeksforgeeks.org/comprehensive-guide-to-edge-detection-algorithms/

4. Sahir, S. (2019, January 27). Canny edge detection step by step in python - computer vision. Medium. https://medium.com/towards-data-science/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

5. Chakraborty, D. (2021, September 29). OpenCV contour approximation. PyImageSearch. https://pyimagesearch.com/2021/10/06/opencv-contour-approximation/

6. GeeksforGeeks. (2024, July 25). Line detection in python with opencv: Houghline Method. https://www.geeksforgeeks.org/line-detection-python-opencv-houghline-method/

7. Goal. OpenCV. (n.d.). https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html

8. *Administering GitHub Actions - GitHub Docs*. (2025). GitHub Docs. https://docs.github.com/en/actions/administering-github-actions

9. game-ci. (2024, May 7). *GitHub - game-ci/docker: Series of CI-specialised docker images for Unity*. GitHub. https://github.com/game-ci/docker

10. Technologies, U. (2024). *How to run automated tests with the Unity Test Framework | Unity*. Unity.com. https://unity.com/how-to/automated-tests-unity-test-framework

11. Technologies, U. (2025, April 16). *Unity - Manual: Unity User Manual*. Docs.unity3d.com. https://docs.unity3d.com

12. *Petru Potrimba. (Sep 25, 2023). What is R-CNN?. Roboflow Blog: https://blog.roboflow.com/what-is-r-cnn/*

13. *Convolution*. (2015). NVIDIA Developer. https://developer.nvidia.com/discover/convolution#

14. *CS1114 Section 6: Convolution*. (2013). https://www.cs.cornell.edu/courses/cs1114/2013sp/sections/S06_convolution.pdf

15. Boesch, G. (2024, August 5). *Faster R-CNN: A Beginner's to Advanced Guide (2024) - viso.ai*. Viso.ai. https://viso.ai/deep-learning/faster-r-cnn-2/

16. Buhl, N. (2023, September 12). *Image Thresholding in Image Processing*. Encord.com. https://encord.com/blog/image-thresholding-image-processing/

17. Mahajan, A. (2023, October 29). *EasyOCR: A Comprehensive Guide*. Medium. https://medium.com/@adityamahajan.work/easyocr-a-comprehensive-guide-5ff1cb850168

18. Wang, S. (n.d.). *Connectionist Temporal Classification (CTC) with application to Optical Character Recognition (OCR)*. Retrieved April 20, 2025, from https://cseweb.ucsd.edu/classes/wi19/cse291-g/student_presentations/CTC_OCR.pdf

19. Pranjal Srivastava. (2017, December 23). *Essentials of Deep Learning : Introduction to Long Short Term Memory*. Analytics Vidhya

20. *OpenCV: Morphological Transformations*. (n.d.). Docs.opencv.org. https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html

21. OpenCV. (n.d.). *OpenCV: Canny Edge Detection*. Docs.opencv.org. https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

22. Goyal, D. (2020, April 7). *Perspective Transformation in Python — on Live Video*. Medium; Analytics Vidhya. https://medium.com/analytics-vidhya/perspective-transformation-in-python-on-live-video-bca558876e8

23. *Thresholding, Edge, Contour and Line Detection with OpenCV*. (2022, June 13). Rubix Code. https://rubikscode.net/2022/06/13/thresholding-edge-contour-and-line-detection-with-opencv/

24. *Thresholding-Based Image Segmentation*. (2023, January 11). GeeksforGeeks. https://www.geeksforgeeks.org/thresholding-based-image-segmentation/

25. *Perspective Transformation - Python OpenCV*. (2020, February 5). GeeksforGeeks. https://www.geeksforgeeks.org/perspective-transformation-python-opencv/

26. *Deep Learning | Introduction to Long Short Term Memory*. (2019, January 16). GeeksforGeeks. https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/

27. *Residual Networks (ResNet) - Deep Learning*. (2020, June 3). GeeksforGeeks. https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/

28. *Introduction to Convolution Neural Network*. (2024, March 14). GeeksforGeeks. https://www.geeksforgeeks.org/introduction-convolution-neural-network/