Graduation report

Procedural RGBD pear image generation

Pepijn Wasser

Saxion University of Applied Sciences
Graduation 2022/2023
482906@student.saxion.nl

Abstract

In this report you can read about the pear simulator developed for Riwo as part of my graduation assignment. It gives a description of the company and the end user. It also goes through the design phases, testing, and obstacles throughout the project. Finally it will go into detail about the automated annotation used to train neural networks created as an extension of the project.

Author keywords:

Mesh, Procedural, Pear, RGBD, ROS2, Simulation, Neural Network, annotation, Substance3D, OpenCV

Contents

Introduction: 5
Research questions: 5
Main question: 5
Sub questions: 5
Research Methods: 5
Process of approach: 6
User description: 6
Requirements: 7
Process: 7
Learning ROS2: 7
Determining good and bad pears: 7
Determining the best solution to create RGBD images of pears: 8
Determining the best engine for procedural models: 8
Creating a procedural pear: 8
Creating basic shapes: 8
Creating a cylinder: 9
Creating pear flesh: 10
Creating pear stalk: 10
Creating basic textures: 11
Randomizing pear: 11
Creating pear colliders: 12
Creating conveyor: 12
Increasing pear performance: 12
Generating RGBD images: 13
Connecting to ROS: 13
Creating ROS2 messages: 13
Testing ROS2 messages: 14
Optimizing ROS2 message creation: 14
Fixing build issues: 15
Fixing memory issues: 15
Connecting to the Riwo pear packaging software: 16
Generating textures: 17
Generating textures in shader graph: 17
Learning Substance Designer: 17

Generating base colour: 17
Generating Russeting: 17
Generating dots: 18
Creating cuts: 19
Adding general improvements: 19
Creating black spots: 20
Creating Rot: 21
Creating the pear crown: 22
Testing pear visuals: 22
Bug fixing pear mesh: 23
Bugfix line in mesh: 23
Bugfix stretching of top and bottom: 24
Creating automated semantic segmentations: 24
Saving images: 25
Making the user interface: 26
Creating the design: 26
Implementing a save system: 26
Connecting the settings to the preview: 27
Testing the UI: 27
Changes to the UI: 27
Making automated annotations: 28
Converting images to point clouds: 28
Unexpected behaviour: 29
Connecting holes: 29
Increasing performance: 30
Testing the implementation: 30
Implementing OpenCV annotation: 31
Creating documentation: 32
Presenting simulation at TValley Tech Conference: 32
Final test: 33
Conclusion: 33
Recommendations: 33
Reflection: 34
Reference list: 35
Appendices: 37

Appendix 1: Pear packaging component overview.....38
Appendix 2: Empathy map.....39
Appendix 3: Document describing pear quality.....40
Appendix 4: SWOT analysis on possible ways to generate pears.....48
Appendix 5: Example drawings to help understand what a pear is.....49
Appendix 6: Sketch final UI concept.....52
Appendix 7: UI user test results.....53
Appendix 8: Overview LabelMe.....57
Appendix 9 Resulting annotation with errors from first iteration of automated annotation.....58
Appendix 10: Several UML diagrams and documents.....61

Introduction:

Riwo is a company located at the Zutphenstraat 1 in Oldenzaal, that makes software for machines, robots, and autonomous vehicles. One of their projects consists of a robot that scans pears on a conveyor belt, and sorts the pears based on certain quality factors. The robot makes use of a neural network which takes RGBD images created by a camera above the conveyor as input. To assess if the robot is working, Riwo has a small model of the machine in their workshop. Using this machine requires the user to lay a pear on a conveyor, and manually crank a lever to make the conveyor move. To be able to test the different pear qualities, the user needs to have the different quality pears. This is suboptimal as pears rot, so whenever the customer sends a batch of different pears to test with, the software engineers only have a limited amount of time to test the machine. To fix this issue, they tasked me to make a system which replaces the entire physical machine by simulating the RGBD images which would normally be created by the camera above the conveyor belt.

Riwo uses Robot Operating System 2 also called ROS2 to develop their robots. They specifically use the ROS2 Humble distribution. ROS2 uses networked nodes to create complex logic. This allows the system to have interchangeable components. For example, it is possible to switch a camera or sensor without it influencing the detection of objects. See appendix 1 for examples of different software components a robot might have. Most nodes in ROS2 are written in C++ or Python, which means it can include external libraries. Riwo uses the OpenCV package in combination with the YOLOACT EDGE neural network for the detection of pears on the conveyor belt. Another piece of software Riwo uses is Gazebo Ignition. Gazebo is an engine which is used to simulate, and test developed code before it is rolled out in the real world.

I was part of the research and development team, coordinated by my company supervisor Nathalie Geerlings. The team consists out of eight people several of which conduct their internship or graduation. Most team members do not work on the pear sorting machine, but they instead work on other projects including autonomous vehicle development or apple picking algorithm creation.

Research questions:

Main question:

Is it possible to create RGBD images of pears in a game engine, and have the generated images be connected to the ROS2 network so that developing a pear sorting robot becomes more efficient?

Sub questions:

What makes a good or bad pear?

What ways are there to create RGBD images of pears?

How can one send the generated data to ROS2?

What engine is best suited to create RGBD images of procedural pears?

Research Methods:

By researching and implementing all the sub questions, we can answer the main question thus, one needs to define how to approach the sub questions to properly get answers to their questions.

To get to know what makes a good or bad pear, Riwo already had contact with the customer. Unfortunately, all this information is scattered across PowerPoints, system files and pieces of code or mental notes, meaning that I will have to gather this information, and condense it to a document which one can use as an anchor point in developing the pear errors.

It is important to determine the way one wants to generate the RGBD images before one selects an engine because, the way images are generated may use various aspects of an engine like the physics

engine or image processing tools, making certain engines better in different scenarios. To get to know what solution I will use to generate RGBD images, I plan to do online research into what solutions are possible and make a SWOT analysis to determine the solution I will develop.

If one manages to create RGBD images, the images will need to be sent to ROS2. If I were to use a game engine like Unity or Unreal Engine, I would need to research how to generate ROS2 messages which contain the data of the generated images, and how to send them to the software developed by Riwo. To get to know this, I plan to look at the ROS2 and OpenCV documentation and the documentation of the package I will use to link to ROS2, and the software developed by Riwo.

Riwo normally uses Ignition Gazebo for their simulations, thus it would be beneficial if I would also use this piece of software however, depending on the way I generate RGBD images, and how good support for ROS2 is in different engines, other solutions might be more suitable. Based on online research and experimenting with Gazebo, I plan to choose the most suitable engine.

Process of approach:

With the information we gather from the sub questions, I could gradually make a simulation which generates images of pears. The easiest images to generate are good and straight pears. Thus, at the start, I would focus on generating images of good pears, and testing if the generated images are recognized as good pears by the robot by sending them to a neural network over ROS2. If the images were correctly recognized as pears, it would affirm that my solution works correctly.

After having created good pears, I will generate pears with mistakes. Based on the list of faults a pear can have, I will discuss which errors are most important, and thus which ones I will focus on. Like the good pears, if one were to generate pears with lesser quality and feed the images into the neural network, one can validate that the simulation works.

To make the simulation user friendly and easy to use, I will need to make a graphical user interface (GUI) which gives the users control of the pears being generated. To evaluate if the GUI truly is friendly to use, I will need to conduct a user test, and implement the feedback in an updated version.

User description:

To better understand the user I will be making a solution for, I made an empathy map which can be seen in appendix 2. With this empathy map in mind, I created a user description.

Since most of the people that will use my application are software engineers, they do not necessarily have knowledge about game like mechanics such as save files or advanced user interface options. This also means that they do not have Unity on their workstations.

All the developers use the Ubuntu distribution of Linux as their main operating system. This means my application should work on Linux.

Currently the developers can assess their developed software by manually cranking a wheel to move a pear across a model machine in the workshop. Using this model machine takes a lot of time which would be better spend on developing software.

The application developed by Riwo that scans the conveyor for pears uses a neural network which needs to be trained on a dataset. To create this dataset the developers, need to use the model machine to generate images, and then manually annotate hundreds of these images which can take several hours to days. This again is a timely job which most find boring to do.

Requirements:

Because the actions the user needs to perform mentioned in the user description can take several hours, my solution should be quicker to use than the current workflow. As the users might not be used to save files and an expansive user interface, the application should be intuitive and easy to navigate. Since the users do not have Unity, it is necessary that the application is playable in an easy to install build. As the users use Linux, packages being used should support this platform.

Process:

Learning ROS2:

In accordance with the company introductory manual, and since it is important to know how the company and their software works, I started my graduation by going through the official ROS2 tutorials to learn the basics of ROS2. I saw how ROS2 has nodes, and how the nodes communicate (Understanding Nodes — ROS 2 Documentation: Foxy Documentation, n.d.). I managed to make a robot which can be moved with terminal commands.

Determining good and bad pears:

After having been introduced to the company, my colleagues, and after having learned the basics of ROS2, I made a document describing the pears that will need to be generated. The document, which can be seen in appendix 3. It describes the pears the customer works with, and the possible faults a pear can have. I saw how pears can be wildly misshapen (Figure 1A), miscoloured (Figure 1B), or how pears can be injured or rotten (Figure 1C).

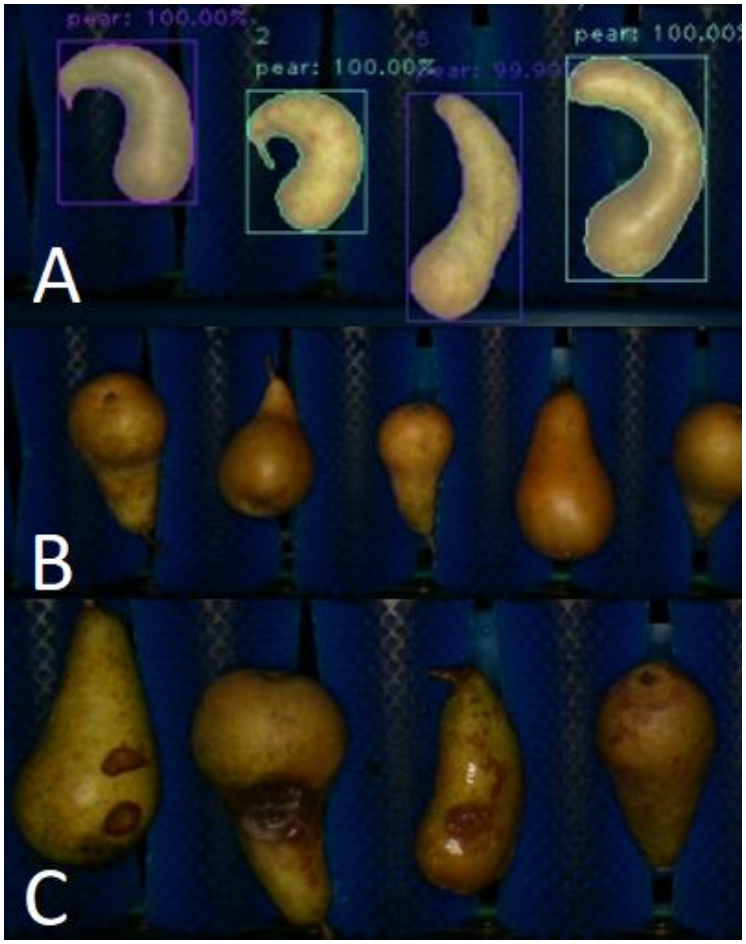


Figure 1 Image showing misshapen pears(A), miscoloured pears(B), and rotten pears(C)

Determining the best solution to create RGBD images of pears:

Having a clear vision of how pears can, should and should not look, I started to determine the best way to generate RGBD images, and devised several ideas. The first idea was to modify existing RGBD images by stretching or warping them to create diverse sizes and shapes. By adding certain colour patterns, I could simulate injuries and colour issues. Another option was to use Houdini to create procedural models based on parameters. The third idea was to train a Generative adversarial image synthesis neural network and have it generate images based on parameters. Lastly there was the option of generating 3D models of pears and have a camera in the scene generate the RGBD images.

I read up on the way these concepts could be implemented, and based on the SWOT analysis in appendix 4, I determined that generating 3D models would be the most suitable method as it doesn't require any datasets, since it gives a lot of control of the exact images and it allows for runtime generation.

Determining the best engine for procedural models:

With a solution defined, I started to investigate the engine I would use. Riwo uses Gazebo Ignition for their other simulations, so I started investigating this engine. According to Aderinola (2019), Gazebo is "A 3D simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments", thus the focus of Gazebo is not on visuals. It also did not allow for complex procedural meshes to be created, and editing the engine was difficult. In the end these issues made me decide against using Gazebo as the engine for my simulation. I also investigated Unity and the Unreal Engine as potential candidates for the simulation. Since I was more familiar with these solutions, I mostly investigated how well their support for procedural meshes was and if there was a good way to connect the engine to ROS2. I learned that Unity has an official ROS2 package, and that it had better support for procedural meshes, so I went with Unity as my engine for the simulation.

Creating a procedural pear:

Creating basic shapes:

I started creating my procedural pear mesh by following a tutorial on how to create a procedural triangle. I learned how the order of the triangle influences what side is being rendered, and how to make buffers (Jayelinda, n.d.). After creating a basic triangle, I made two quads out of four triangles and six vertices (figure 2).

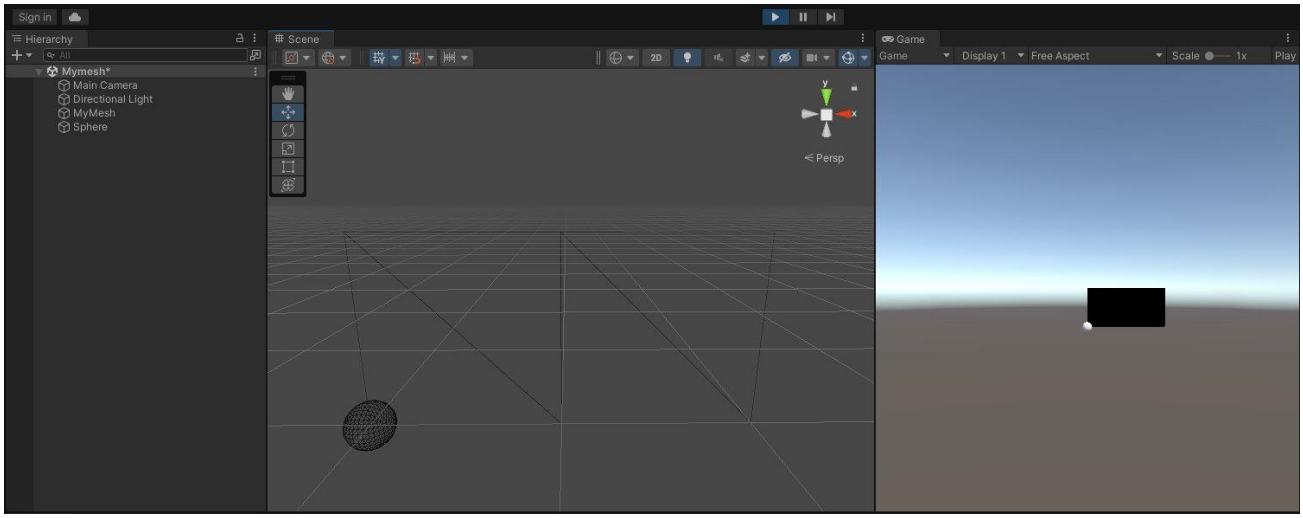


Figure 2 Two quads made up of four triangles.

After having learned how to create a quad, I started to experiment with moving points around and made a triangular prism (figure 3).

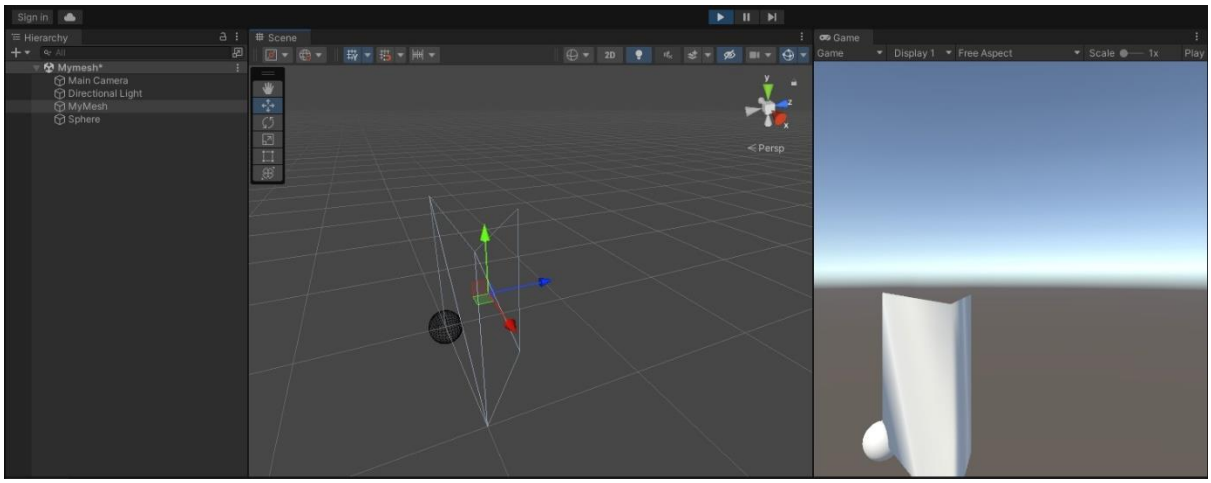


Figure 3 Procedurally generated triangular prism.

Creating a cylinder:

Having gained knowledge about how to create basic meshes from scratch, I started to determine how to make a cylinder. I planned out where I should place the vertices and how to connect them. I determined that it was the easiest if vertices on the same horizontal layer have indexes after each other. I placed the vertices around a centre in a circular shape for a defined number of layers. The number of vertices per layer, and the number of layers could be defined by the user in the inspector.

After having placed the vertices, I needed to connect the right vertices with each other. I saw that for any given vertex that is not the last index on a layer, it should connect to the vertex with the next index, and the vertex with an index equal to the current index added to the number of vertices per layer. If it is the last index, it should connect to the first vertex on the layer and the vertex with the next index. See figure 4 for an overview of the connections. A similar method has been used to create the other triangle that makes up the square at a given index.

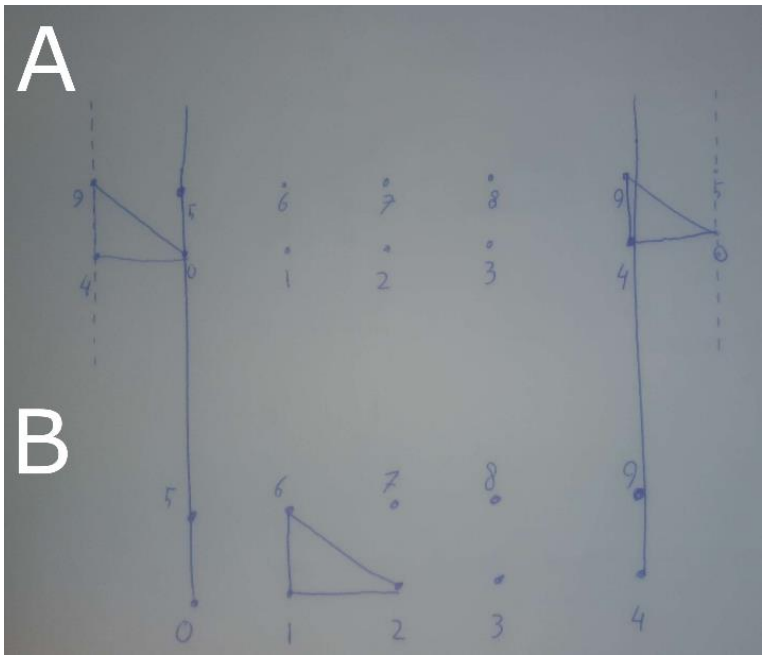


Figure 4 connections with index at end of row (A) or vertex index in middle of row(B).

With all the vertices and connections in place, I managed to create the cylinder as seen in figure 5.

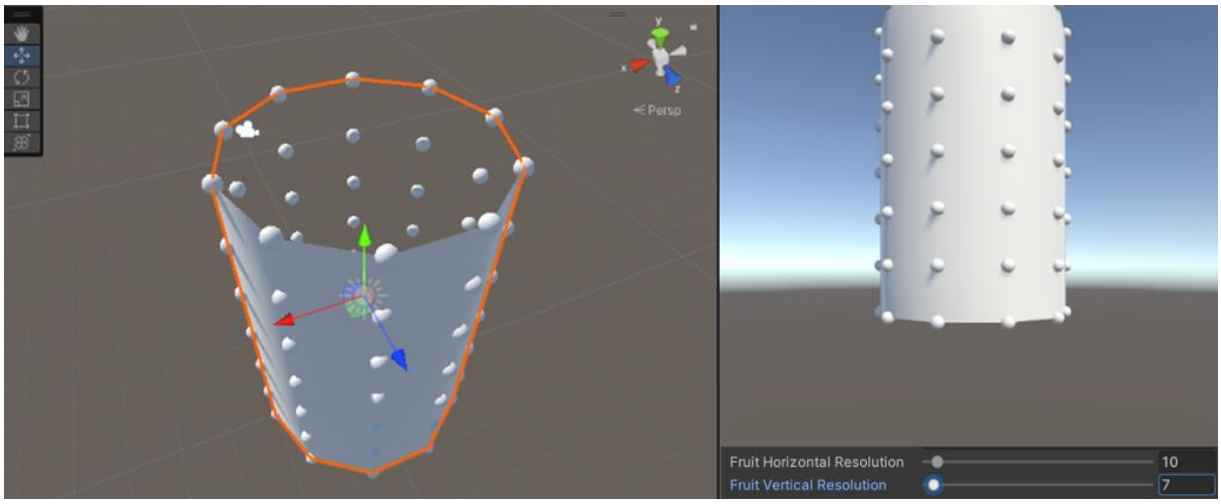


Figure 5 Procedurally generated cylinder with dynamic number of vertices.

Creating pear flesh:

To generate the part of the pear one eats or the "pear flesh'', I had to devise an organized way to place the vertices in a pear shape. I started by analysing the shape of a pear and came up with the following definition: "A pear is a cylinder with varying thickness around a core that can be curved". Before I arrived at this definition, I made several drawings and notes to help me understand a pear. These drawings can be seen in appendix 5. Based on this definition I used the logic of the cylinder in combination with a user defined animation curve to generate the pear flesh. With this implementation the user can set a curve that resembles the offset from the core and define the shape of the pear in an intuitive way.

Creating pear stalk:

A pear also has a stalk sticking out of the top. To make the stalk, I used the same concept as for the pear flesh except that the cylinder does not have a varying thickness rather, it has three animation curves. Each curve represents the rotation of the stalk over time. If one sets the curves to nice looking values, we can achieve results similar to figure 6. The pear flesh was also modified with this logic to make it so the pear can be curved in certain directions.

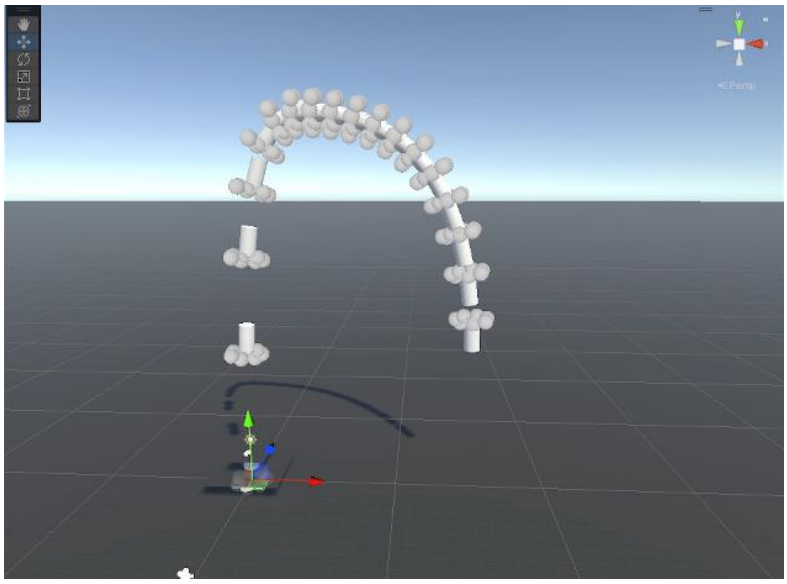


Figure 6 Stalk generated based on animation curves.

Creating basic textures:

Because it is possible to “unfold” a cylinder into plane (see figure 7), it was easy to set the correct UVs of the vertices. We just have to unfold the shape and see where each vertex ends up. With the UVs set, we can apply a wooden texture to the stalk, and a pear texture to the pear. The only issue with this implementation is that the part at the top and bottom where the vertices are almost equal have some stretching, but I decided to postpone this issue as it would possibly not have influence on the simulation.

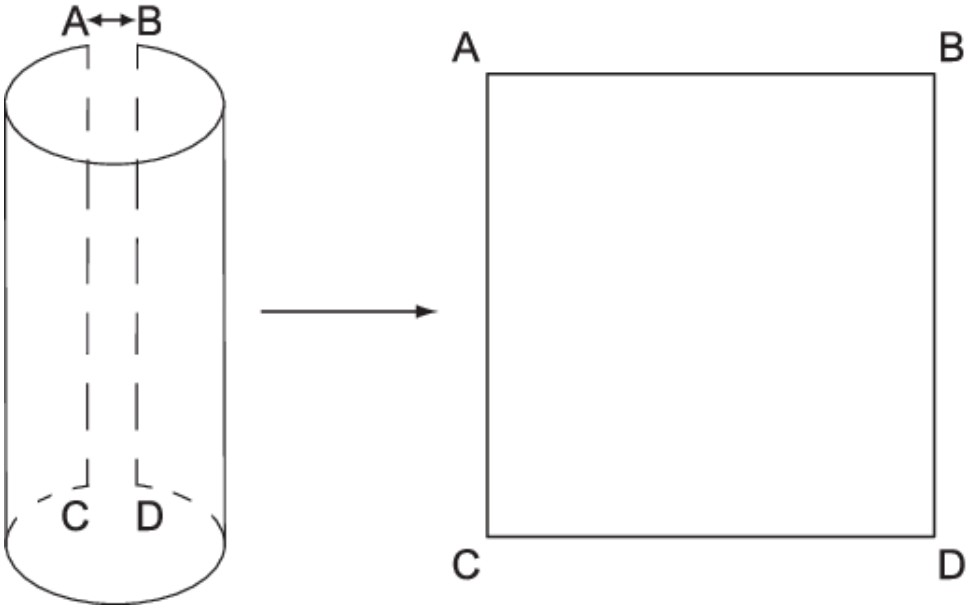


Figure 7 Unfolding a rectangle into a triangle (Druguet et al., 2004)

Randomizing pear:

Now that we can generate a procedural pear based on 7 animation curves (1 for flesh thickness, 3 for flesh direction and 3 for stick direction), we can generate random pears by defining 2 curves in each animation curve, and have the program create a new curve between the two curves we set (Unity Technologies, n.d.-c). To implement this, I changed the animation curves into min max curves. In figure 8 we can see three randomly generated pears.

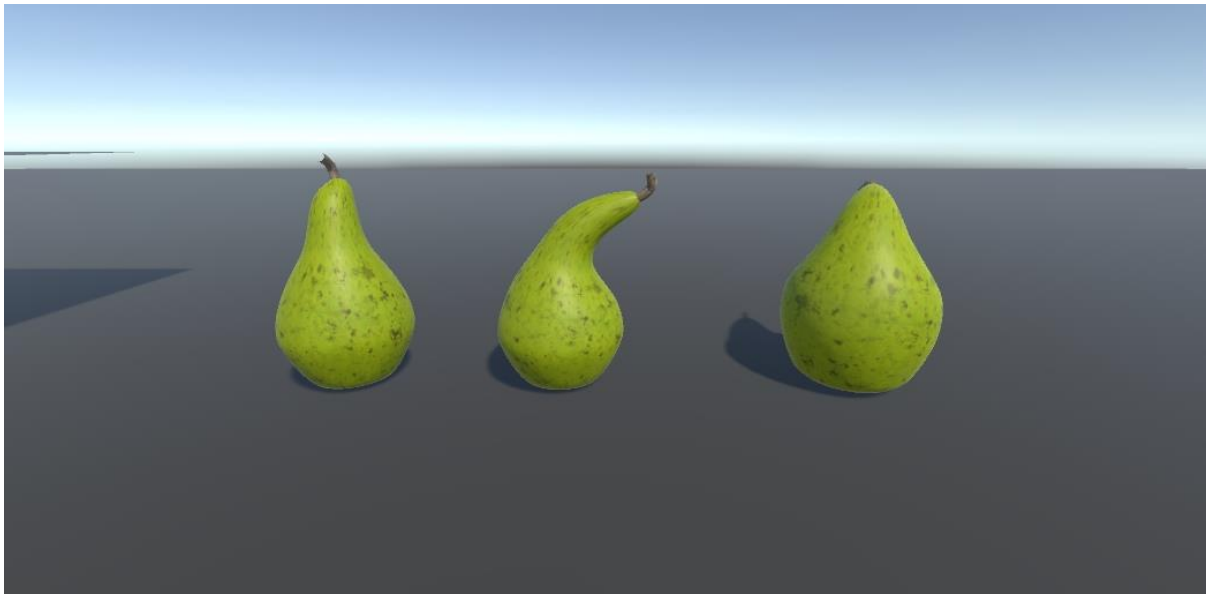


Figure 8 Three procedurally generated pears with random shapes

Creating pear colliders:

When real pears move across the conveyor, they sometimes roll around giving the camera difficulties with tracking the pear instances. To simulate this behaviour, I needed to make the pears behave like rigid bodies. Due to technical limitations, Unity rigid bodies do not support non-convex mesh colliders (Unity Technologies, n.d.-a). Since the simulation needs to be as realistic as possible, non-convex mesh colliders were mandatory. To fix this issue I used a package which generates a lot of cube colliders based on a mesh collider (Sanukin, n.d.). This created an accurate representation of the pear mesh collider that could be used in the physics engine. See figure 9 for the generated collider.

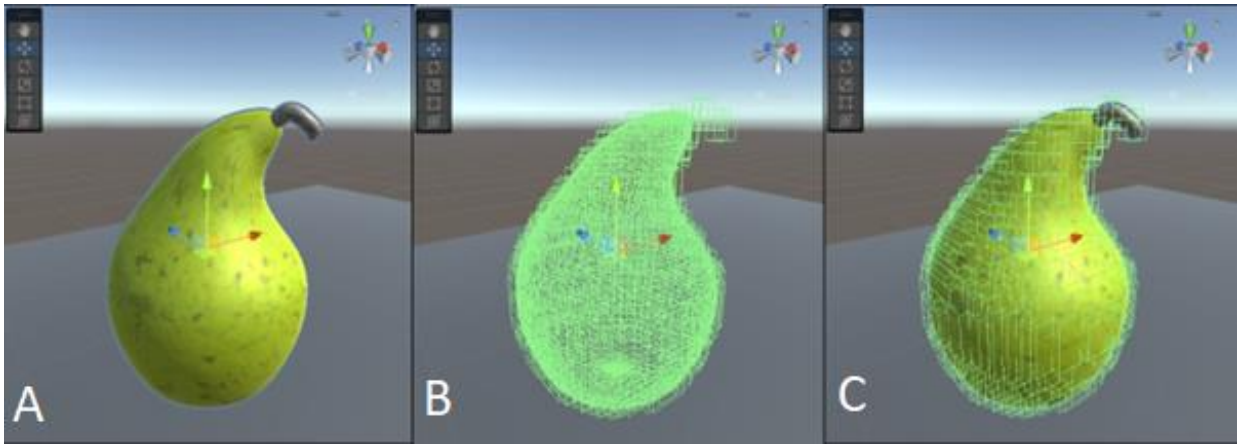


Figure 9 Procedurally generated pear(A), the procedurally generated collider(B) and both combined(C)

Creating conveyor:

After making a script that spawns a pear every so often and after giving the pear a good collider and rigid body, I started to work on a component which moves the pears across the roller conveyor. I ran into the issue that the force I added to the rigid body was not enough to push the pear across the small incline of the roller, and a force that would move the pear across the roller would have too much force so it would fly across the conveyor once the resistance of the roller was gone. To solve this issue, I gave the pears an additional upward force to help get them over the incline of a roller.

Increasing pear performance:

The pear generating and conveyor system was working, but every time a new pear was being spawned, it would massively spike performance (figure 10).

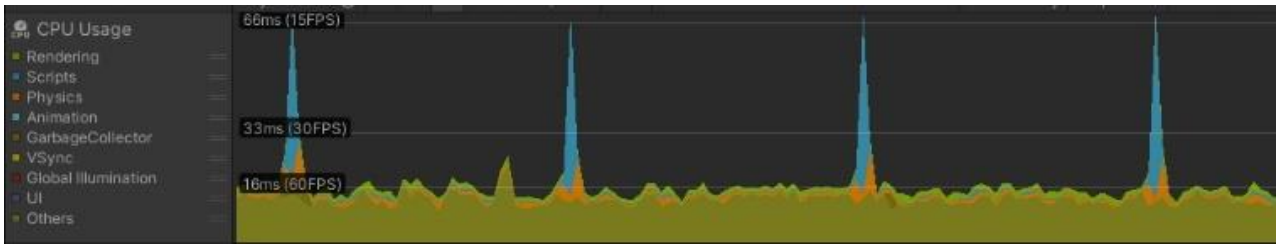


Figure 10 Performance spikes on pear spawn

After debugging the performance using the Unity profiler, I found that generating a collider took a big toll on performance. To fix this, I made modifications to the script so that generating the collider became asynchronous. This divided the process of generating a collider over time, but now colliders were being generated while the pear was already on the conveyor. I decided to make a variation of a pooling system in which spawned pears would move to a buffer of pears when they completed

generating their collider. When the conveyor needs a pear, instead of spawning a new one, it takes a pear from this buffer (Figure 11).

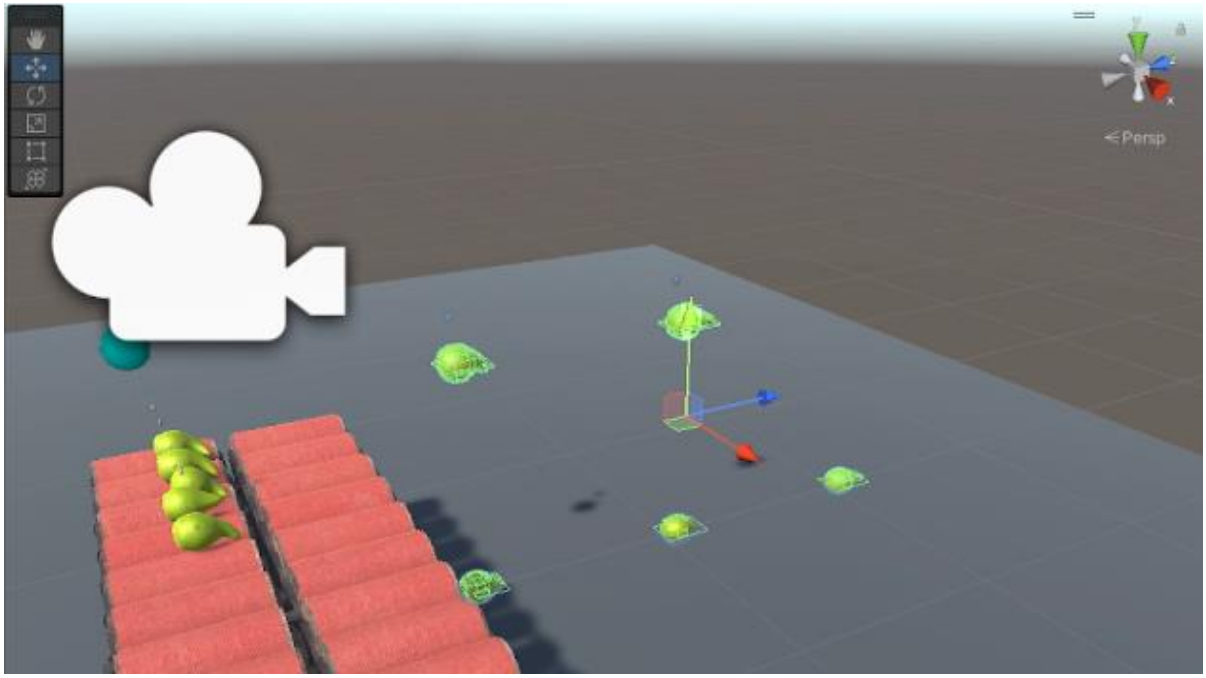


Figure 11 Pears being spawned on the bottom row, and pears in buffer on top row.

When one compares the performance before the changes (figure 10) and after the changes (figure 12), one can see that the big spikes are no longer there. We can also see that the performance is around 60fps even if we spawn a pear every second.

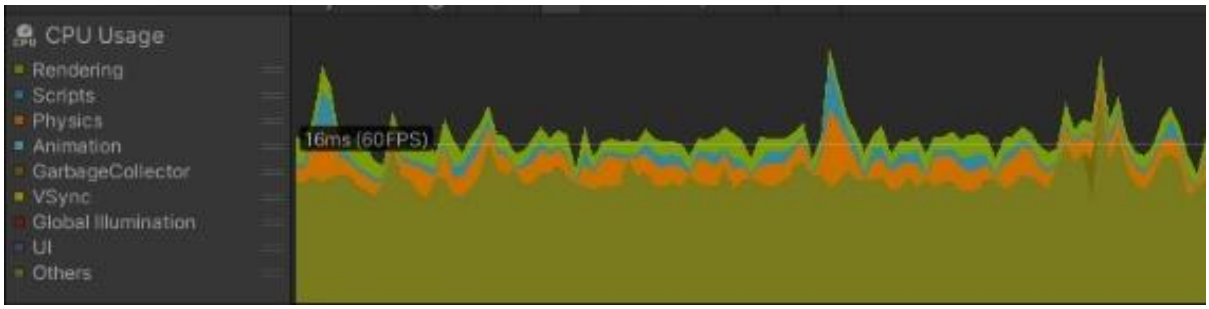


Figure 12 Performance after pooling

Generating RGBD images:

With the pears working, I started to investigate how I was going to create the RGBD images. I came across a GitHub repository called image-synthesis which contained a code package that captured RGBD images. This package also had the potential to capture different image types like a semantic or instance segmentation image. The package made use of command buffers and replacement shaders which I never used before. After researching both topics, I leaned that they were used to render an image with a different shader (Kalathil, 2018). Later in the project I removed this package due to performance reasons.

Connecting to ROS:

Creating ROS2 messages:

When I was researching what engine would be more suitable to implement ROS2 communication, I saw that there were several packages to connect C# to ROS2 like ROSBridge, an open-source

package developed by Siemens called ROS# and an official package developed by Unity called ROS-TCP-Connector. I decided to go for Unity's official package as ROS# made use of ROSBridge for its communication, and Unity's official package was built on ROS#, thus I determined that using the highest-level package would likely have the most features. The ROS-TCP-Connector also had more debugging tools and tutorials than ROS#.

I followed the tutorials that came with the package to see if this package was suitable for the things I needed it to do. I managed to make a cube change colour by receiving terminal commands, and have it broadcast its position back to the ROS2 node in the terminal. After completing these tests, I determined that the package was suitable for the ROS2 communication.

I cloned Riwo's repository that receives the RGBD camera data and determines the pear quality to see what the required content of the ROS2 message was. I learned that the message required an encoded image as well as image descriptions. Since a RGBD camera is a common sensor, the package already contained a way to initialize a message, but I had to initialize it with the correct values. The message also required information I had never heard of like header stamps or a big-endian bool. After discovering that big-endian influences the order of stored bytes, I managed to send the images to ROS2 (Bedell, 2021).

Testing ROS2 messages:

To assess if the messages were created properly, a colleague recommended that I send the message to a program called Rviz2 before I would send them to Riwo's software. Rviz2 is a graphical user interface that watches for ROS2 messages on the network. It also counts the messages received and the time it has been active. By using this program, it was easy to detect and debug my messages and the communication between Unity and the ROS2 nodes.

Optimizing ROS2 message creation:

With the messages working, I noticed that the frame rate took a big dip (figure 13). In the profiler I noticed two sources of the performance issues. The first was the way the images were being captured, and the second one was the creation of the ROS2 messages. I fixed the first issue by creating my own implementation of a RGBD camera, instead of using the external package. What makes my implementation better than the package is that I render the screen with one camera instead of once for every type of image, saving several render calls every frame.

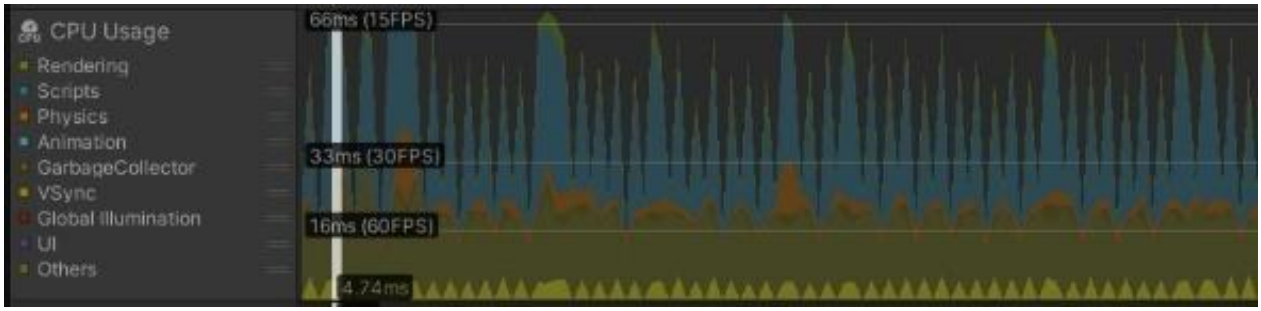


Figure 13 Performance spikes when sending data.

To fix the second issue, I took another look at my code, and noticed that a method called ReadPixels was bad for performance. This method copies the pixels from the render texture generated by the camera into a texture2D, which in turn can be used to access the data itself which is being send in the ROS messages. In my research for a better solution, I stumbled across the function GPUAsyncCallback. It uses the data stored in a more efficient way thus, the function could be used to replace ReadPixels function. After some more benchmarking, I noticed that it was however not better for performance, thus I went back to the ReadPixels method because the GPUAsyncCallback caused stuttering unrelated to fps on the receiving end, and because the performance was slightly

better. In figures 14 and 15 we can see the minimal performance difference between ReadPixels and GPUAsyncCallback. Due to the program still hitting the target 30 fps, I decided to leave this issue for later.

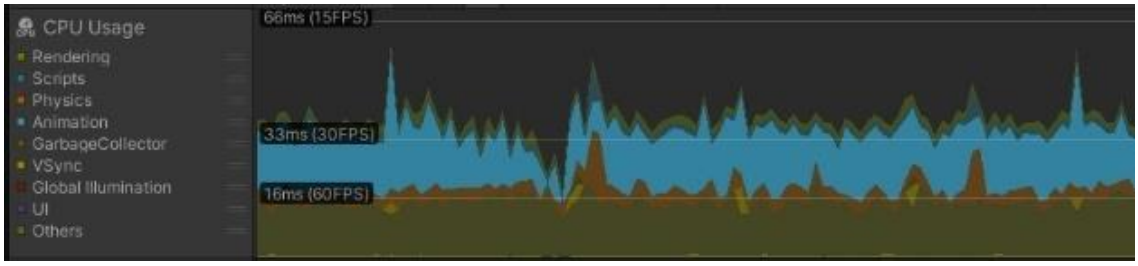


Figure 14 Performance with Read Pixels().

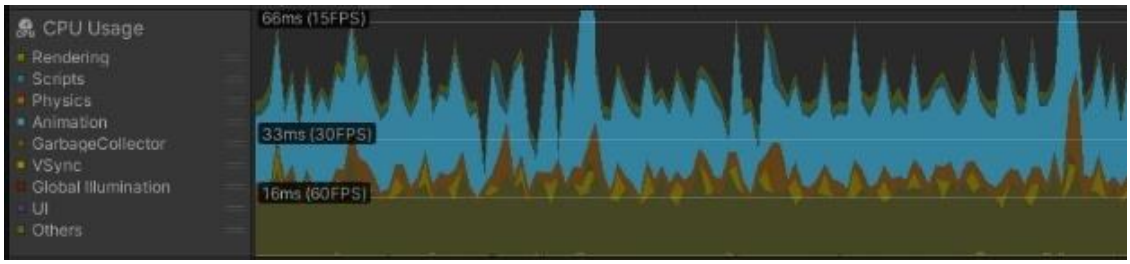


Figure 15 Performance with GPUAsyncCallback().

Fixing build issues:

As I wanted to see the performance of the application without the editor affecting performance, I made a build. When the build was finished, I noticed that my depth texture was not being rendered by the camera. After a day of debugging, I managed to find that a method called Blit was not being called in the build. In this method I cast the view of the camera to a render texture with a special shader. In my research I found that the method does not properly pass the depth values into the shader (Unity Technologies, n.d.-b). To fix this I edited the shader I used in a way that forced unity to write to the Z-buffer. This managed to fix the issue.

Fixing memory issues:

I noticed that there were still performance spikes. In the profiler I saw that it was related to the GetRawTextureData method call in my ROS2 communication script. Uncommenting this line fixed the issue, but it meant we were not sending an image of the generated pears, instead it was a black image. Another thing I saw in the inspector was that the memory would go up in steps, and every 10 seconds it would quickly go down again (figure 16). The moments of this also aligned with the moments in which the inspector indicated that the GetRawTextureData was causing issues.

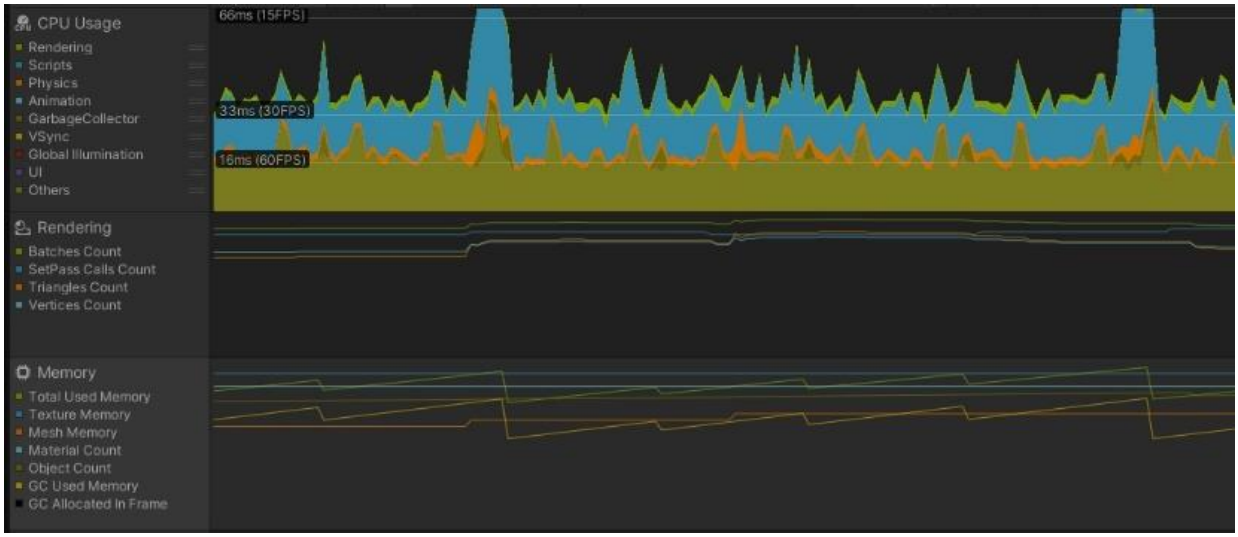


Figure 16 Performance spikes match with memory clean up.

GetRawTextureData was getting the byte array of a texture, so the byte array could be sent to ROS2. We are calling this method thirty times per second, so if one calculates the memory we assign in this method alone, it is already 0.15 Gigabyte per second. In my research I found that the Unity garbage collector likely couldn't keep up with this amount of generated data, so once it got overwhelmed, it would do a more inefficient flush of the created garbage. This is the moment which one can notice every 10 seconds.

As I experienced this issue in the editor only, which is likely due to the editor and profiler also storing the data to display in the inspector and graphs, and since I couldn't find a better way to get the bytes of a texture, I decided to put the issue on hold.

Connecting to the Riwo pear packaging software:

After creating the RGBD camera communication, I installed the software Riwo developed to analyse pears on a conveyor. After setting the message names of my messages to the names the software expects, I managed to get their software to work as can be seen in figure 17. The figure shows a purple bounding box around the generated pears which indicates that the software was recognizing my pears. It correctly identifies the parts of the pear itself, but it also misidentified the space between the roller rows as part of the pear. After discussing the results with my supervisor, it became clear that Riwo's software was not good enough to correctly identify parts of the pear, but that it was good that the pear was being recognized.

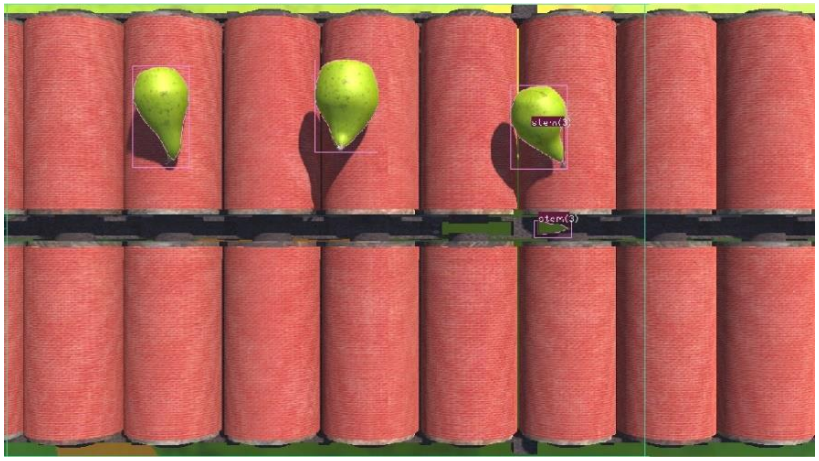


Figure 17 Procedural Pears being recognized by Riwo's software.

Generating textures:

Generating textures in shader graph:

Up until this point my pear texture was the same for every pear, so I started to texture the pears in a procedural manner. I looked at how other people textured procedural meshes, and did not find many sources that did this, so I had to experiment myself. I started with trying to add a cut texture to the existing texture in Unity's shader graph, but the cut was too big as both textures had the same size. This meant I had to use a smaller image of a cut however, pasting images with varying sizes on top of each other took quite a bit of time.

Learning Substance Designer:

Because creating textures in shader graph proved to be quite time consuming and difficult for a simple texture, I started to search for a better option. I knew that Substance Designer was a suitable tool for making procedural textures while exposing settings to Unity. After testing the tool, I determined that this was a way better tool than the Shader Graph, so I started learning this tool.

After watching three basic tutorials on the YouTube channel of Substance Designer, I got the hang of the basics and started to work on my texture. One thing I noticed while making my texture was that many tutorials were for Substance Painter rather than Designer, even if the search specifically mentioned Designer. This meant that I had to implement most of my texture myself.

Generating base colour:

I started by creating a texture which follows a gradient to switch between two colours from bottom to the top. After learning how to expose variables to Unity, I managed to get the texture to work. I expended the base colour by modifying the gradient with noise to make it feel more natural.

Generating Russeting:

A major feature of a pear are brown areas also called russeting. Russeting is not bad for consumers and does not influence the quality however, the robot that analyses the pears can mistake the russeting for rot, so it was important to get this right. Between pears the amount of russeting varies wildly, as does its position on the pear, but it is more prominent at the bottom of the pear.

I used different noise functions together with gradients to make "islands" of russeting, which were denser at the bottom. In figure 18 one can see the generated grayscale image. The black parts indicate russeting while the white parts had the base colour of the pear. Applying this texture resulted in the pear in figure 19.

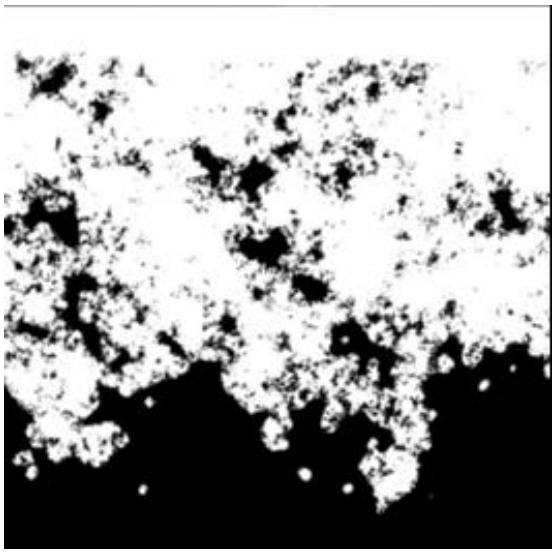


Figure 18 Grayscale image of brown areas.

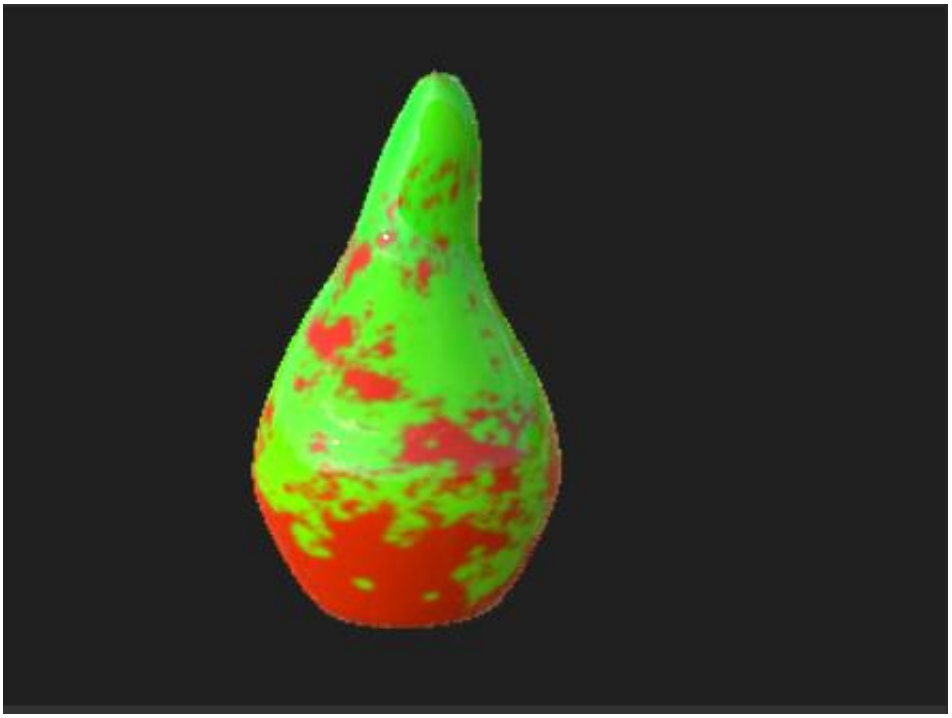


Figure 19 Pear with brown areas (indicated in orange).

Generating dots:

When looking at a pear one can see small dots ranging from green to brown. Some areas have a higher density of dots compared to other areas. Dots however never appear on brown areas. I started by placing dots on the white areas from figure 18 to prevent dots from spawning on brown areas. You can see the dots in figure 20.

After testing the implementation, the generated texture felt quite flat, so I made the brown areas and dots slightly lower than the surrounding surface using a normal map, as to better reflect a real pear.

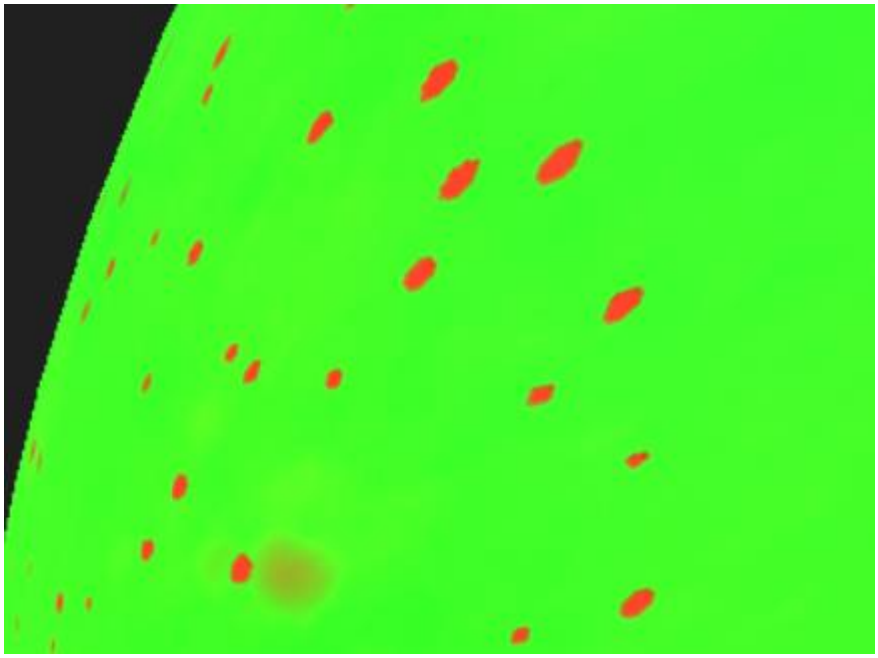


Figure 20 Pear dots generated.

Creating cuts:

Another important part most pears have is some surface cuts which have discoloured brown. I started by creating straight lines and indenting the pear surface on the cuts with a normal map. This felt quite artificial however, so I had to create a way to bend the straight lines. In my research I found out about a node called directional warp. I used this node together with noise to bend the lines in a randomized way. In figure 21 you can see the result.

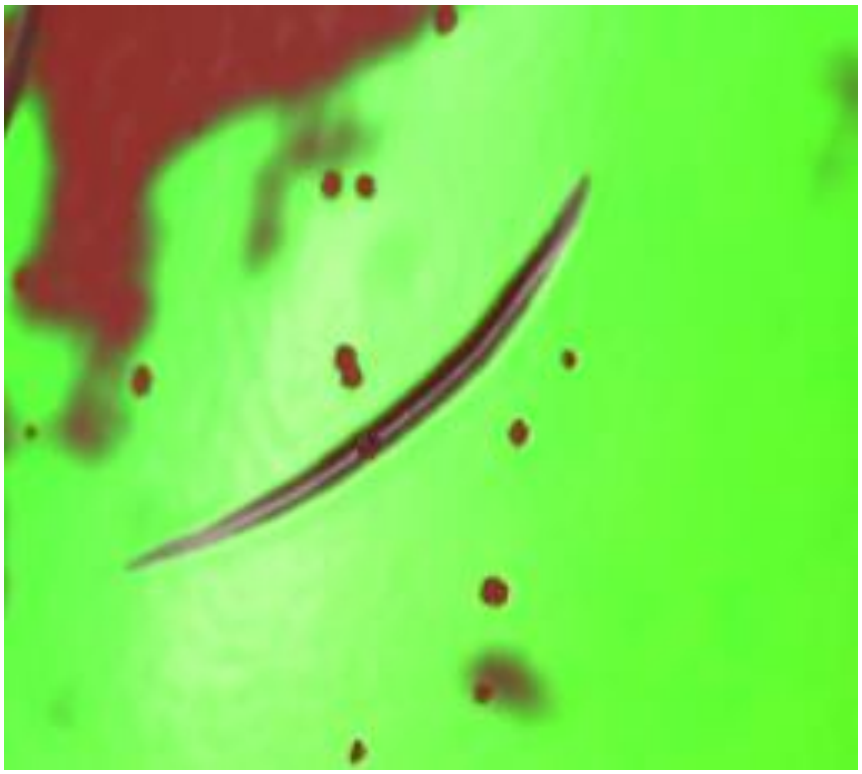


Figure 21 Surface cut.

Since I did not know if the lines were being generated with the correct settings, and since the ultimate goal was to randomize the pears, I exposed several variables to Unity like scratch size and amount. I did the same for relevant settings in the dots and russeting generators.

Adding general improvements:

As the most core features of a pear texture were now in place, I looked at the pear and concluded that the pear looked more like a plastic toy rather than a real pear. I determined that this was due to a lack of detail, so I started to expand on the parts I already made. After looking at a real pear under different lighting conditions, I saw that the brown spaces consist of areas of two slightly different colours, one of which is more reflective. Furthermore, I also noticed that there are some small black dots on the russeting. I implemented these findings and found the result more realistic.

I also determined that the brown dots have a slightly darker colour around them as opposed to the surrounding base colour. I implemented this feature by scaling the dot size, giving that a darker colour, and then placing the dots on top of the slightly bigger area.

Lastly, I gave the base pear material more detail by adding more noise and giving the overall texture a slightly randomized roughness. The results of these changes can be seen in figure 22.

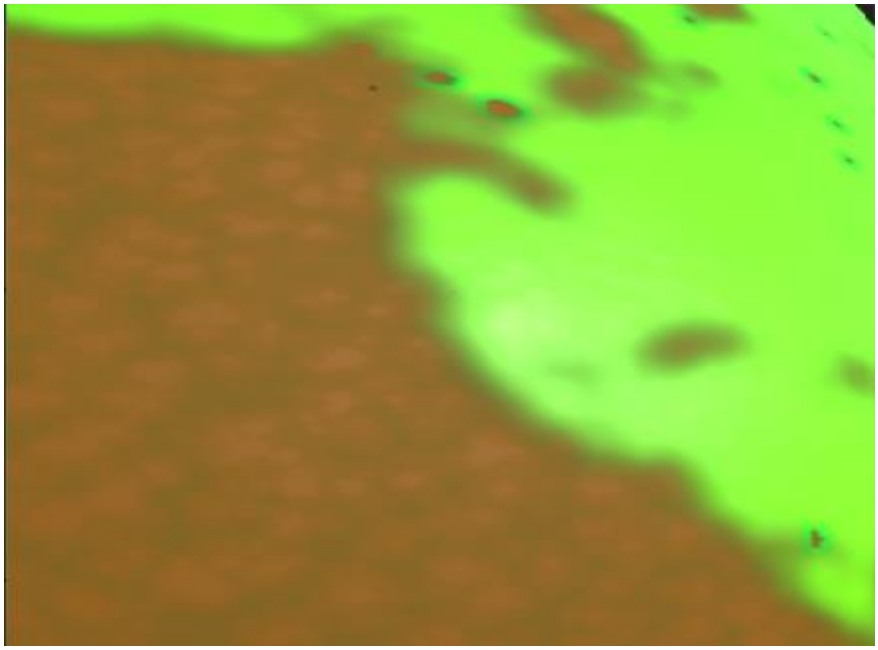


Figure 22 Texture with more reflective black parts, Dot borders and a rougher base shape

Creating black spots:

Pears sometimes also have black areas of little dots so densely packed together that they are almost indistinguishable from one another. These were easy to implement as Substance Designer already had a noise generator which looked like the result I wanted. In figure 23 you can see the black spots, and in figure 24 you can see the current pear in Unity.

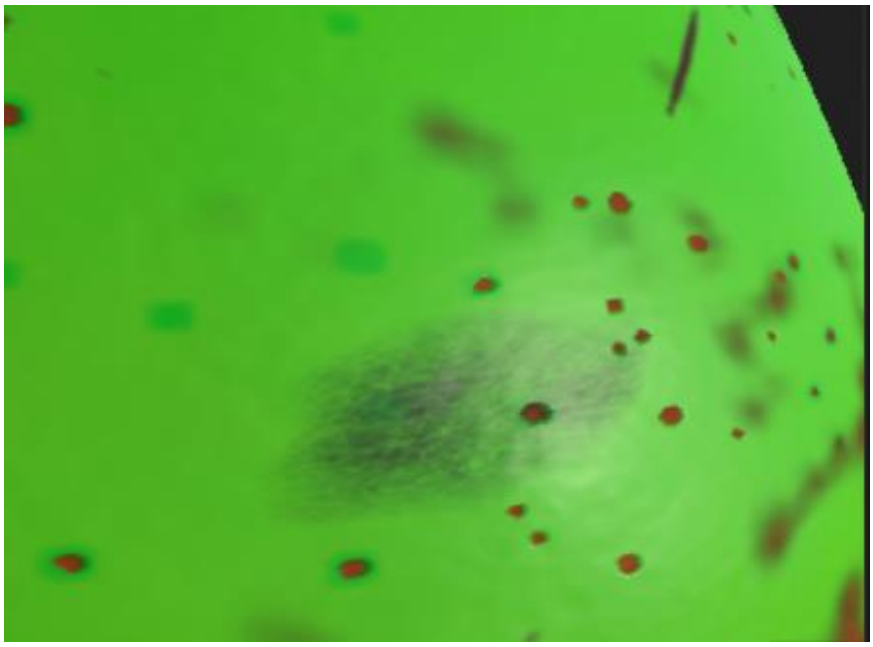


Figure 23 Black spots.

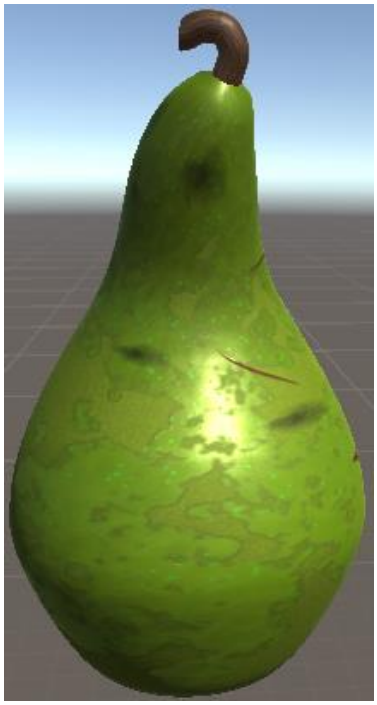


Figure 24 Pear with colours set to green rendered in Unity.

Creating Rot:

The last feature I had to add was rotten areas on the pear. Because there are distinct types of rot, I decided to make a document explaining the several types of rot. Based on this document I started by creating circular areas with a shiny brown colour. After judging the looks of this, I determined that it would increase realism if I added mould to the texture. The result can be seen in figure 25. After a weekly meeting with my company supervisor, we determined that the looks were good enough, and that I should focus on creating other parts of the pear.

Figure 25 Pear texture with rot and mould.

Creating the pear crown:

I decided to continue by implementing the crown of the pear on the bottom. The reason for this was that the programmer of the neural network was working on rot detection, and they were having issues with the neural network detecting the crown as rot, which has a major impact on the pear quality. After looking at pear crowns and discussing the requirements with my supervisor, I determined that the crown does not need to be perfectly accurate, and that several smartly placed triangles would do the trick (figure 26). These triangles are placed in a small indent most pears have. Creating this indent was quite easy as I just had to increase the y value of the core, which was already implemented with my core bending curves.

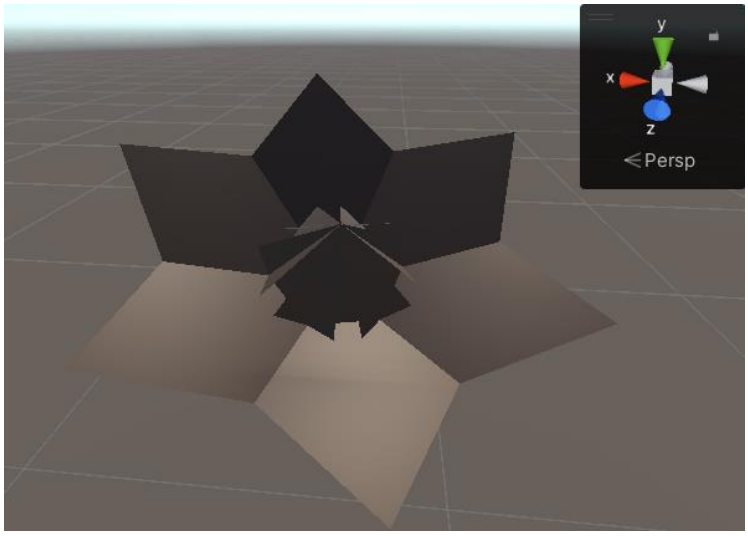


Figure 26 Pear crown made up of several triangles.

Testing pear visuals:

Having created a model that I thought resembled a pear quite well, I conducted a small test in which I cut out a generated pear from a screenshot and edited it into a real image without altering the colours. The created image can be seen in figure 27. I than showed it to both colleagues and friends. Initially they questioned why I showed them that image, but after pointing out that there was a fake pear among them, they quickly noticed the pear. They mentioned that the points that gave it away were mostly the lighting on the edges of the pear and a slightly different pear colour. However, the fact that they were initially unimpressed indicated that my pears were looking quite realistic.

Figure 27 Generated pear indicated with arrow photoshopped in between real pears.

Bug fixing pear mesh:

Bugfix line in mesh:

Up until this point my pear mesh always had an issue I ignored. There was a small seam were the sides of the tiling texture connected caused by the pixels interpolating the colours between the last vertex of a row and the first vertex of the next row (Figure 28). After trying various fixes, I settled on placing the last vertex of each row on the first vertex of the row. This had one issue however as the lighting did not properly work on this edge. From the 3D rendering course, I remembered how vertices on the same position need to average the tangent between the two vertices on the same spot and their neighbours, so after implementing this interpolation, the seam was gone (Hahmann & Bonneau, 2008).

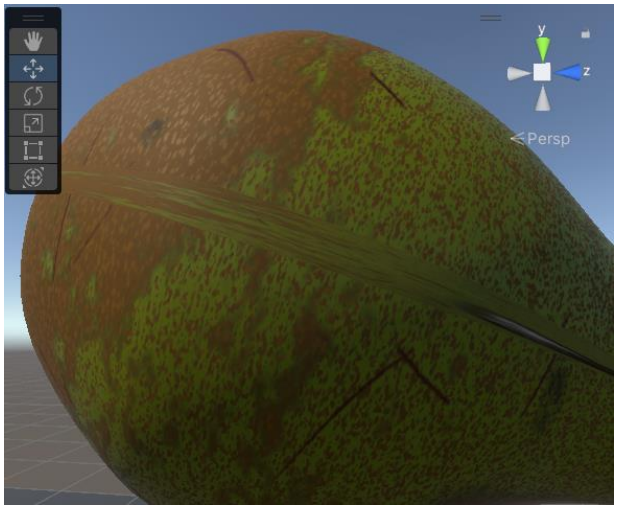


Figure 28 Seam on in the pear mesh

Bugfix stretching of top and bottom:

Another small issue in my mesh was in the distribution of the vertices relative to the height of the pear. Until now, I assumed that my pear was a cylinder, so I positioned the vertex layers evenly. This is however not an even distribution if we consider that various parts of the pears have a different width. If the width of the pear quickly changes like in the top and bottom, this uneven distribution causes stretching (figure 29).

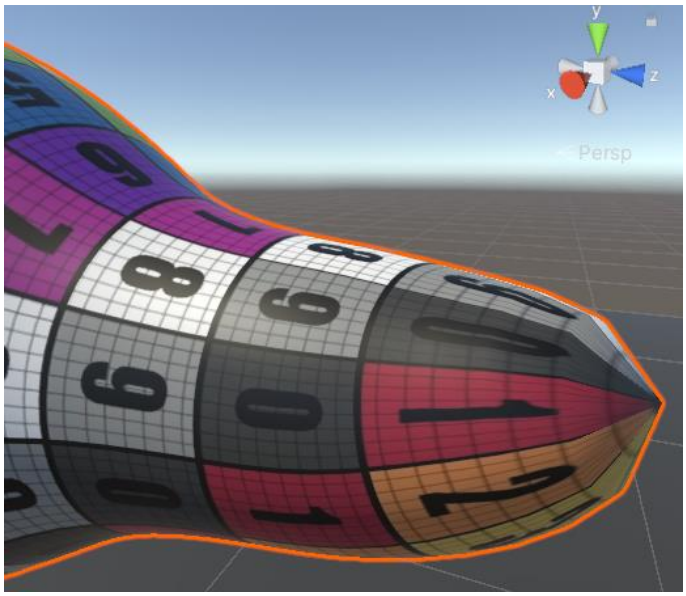


Figure 29 Stretching UVs due to uneven vertex distribution.

To be able to fix this issue I would need to calculate the circumference of the pear before placing the vertices. This is however very costly in performance and mathematically difficult. To solve this issue, I reworked the animation curves I had used into my own custom curves. Just like in the old system the user can define his own animation curves, but based on this curve we create a new curve which places and connects the points with straight lines. This is implementation is a faster way of approximating the circumference. Now when we want to know the width of the pears at a certain height, it bases the width on the approximate circumference curve. This made the vertex distribution more evenly and fixed the issue of the textures stretching.

Creating automated semantic segmentations:

As my graduation assignment was done except for the UI, and since there was still enough time left, me and the company supervisor decided to expand the assignment by implementing automated semantic segmentation. These segmentations create areas of solid colours based on features in an image. This allows them to be used to train neural networks. Since my substance file has the ground truth of the texture, I created a variation of the output texture in which I replace all the details, like rot or scratches, with a solid colour. After learning how to create multiple materials in Unity based on a single graph, I got the images working. Figure 30 shows a segmentation image and its source texture.

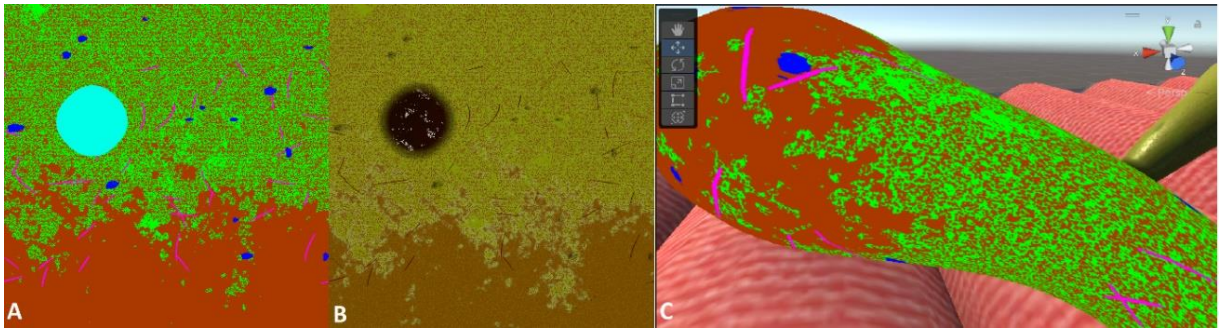


Figure 30 Semantic segmentation image (A), original texture (B) and segmentation texture applied (C).

In Unity, an object can only have one material, but we want to render the object with the colour texture and with the segmentation texture. To solve this issue, I swap the material of the object after the scene has rendered, I render the scene a second time. This solution was working, but it decreased performance as we now render the scene an additional time.

Saving images:

Since the neural net feedback image is only used when creating a training dataset for the neural net, I did not need to send it over ROS2, but instead I had to save the files locally. I implemented a system that saves the images in a folder with the current data, which has subfolders for the different images like RGB and the feedback images. The images were named based on a unique index. This way it is easy to find the matching RGB and feedback images.

When looking at the generated feedback image however, I noticed that the solid colours of the various parts of the pear were blending at the borders of the colours causing a slightly blurred image (figure 31). Since this makes the images useless for training a neural network, I had to find the source of this problem.

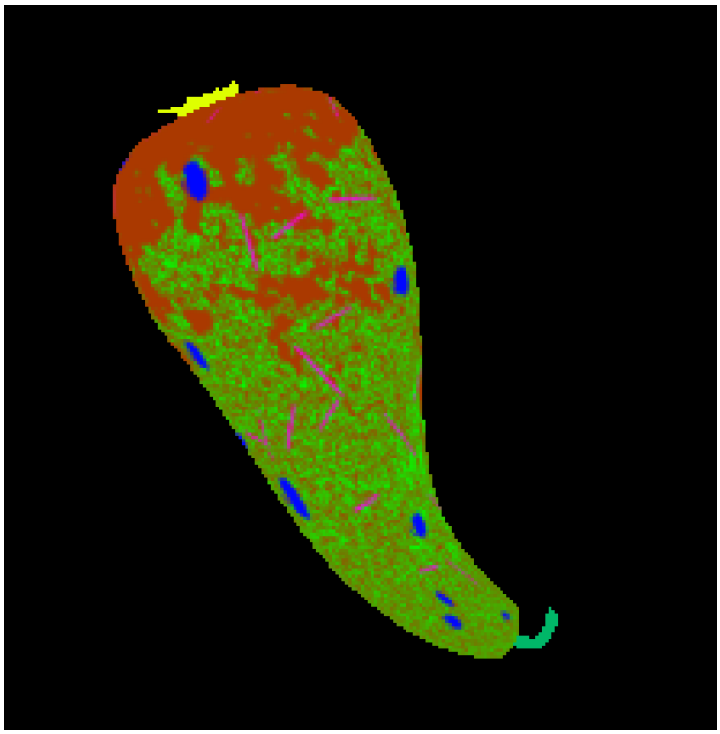


Figure 31 Feedback texture with blurred pixels

I noticed that the blending would mostly go away when increasing the image size to something massive like 12800 x 7200. This is however not a suitable solution as this decreases performance. I also tried to change texture and camera settings like anti-aliasing or MSAA, but this did not have a noticeable effect. In the end I found out that the Mipmaps generated by the substance graph were the cause of the problem, so after disabling the mipmap generation, the images were being generated correctly in a normal resolution (figure 32).

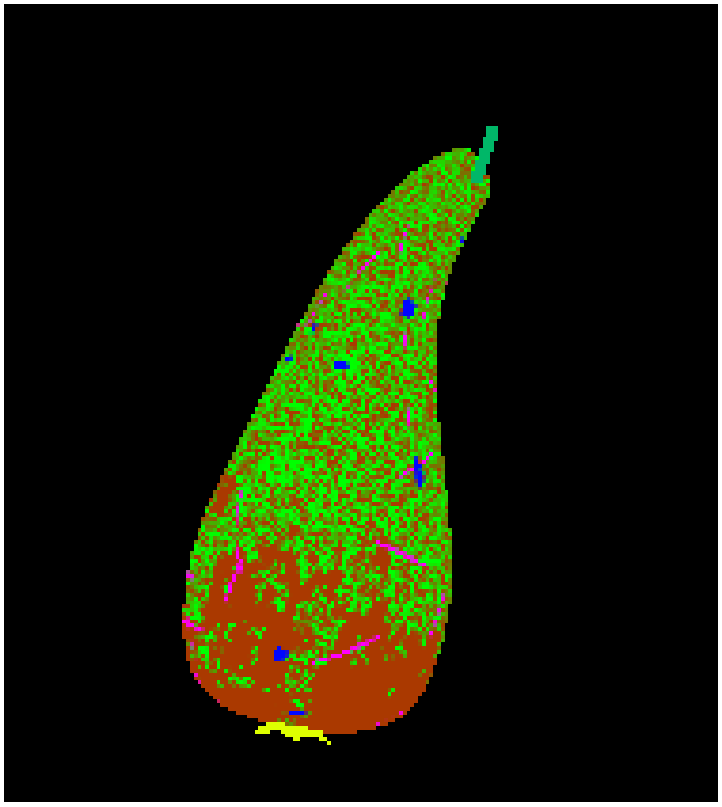


Figure 32 Feedback image with sharp pixels (any blending is due to image scaling)

Making the user interface:

Creating the design:

With the functionality for creating a neural network dataset done, I determined that starting with the user interface would be my best course of action as this would allow my colleagues to start using my tool, and since it would allow me to conduct an early user test. I started by determining what needs to be in the UI, and followed that by creating several sketches. I asked my colleagues which they deemed most suitable and started to implement that design in Unity. It was settled that the UI would contain a preview of the pears with options around the preview sorted in sections. In appendix 6 you can see the sketch of the concept.

Implementing a save system:

When discussing the UI with my company supervisor, it became clear that an important feature of the UI would be the option to load settings from an external file. I settled on a system that loads a JSON file in which all settings have their values are stored. When the user is using my UI, they can make changes which can be saved to the JSON file, or which can be for that session only. This implementation allows the user to swap out the JSON file or edit it with a text editor. In figure 33 you can see two example JSON files.

SaveData - Kladblok

Bestand Bewerken Opmaak Beeld Help


```
{
    "toggleDict": {
        "items": [
            {
                "key": "SaveDepthImage",
                "value": false
            },
            {
                "key": "SaveNeuralNetGrayscaleFeedbackImage",
                "value": false
            },
            {
                "key": "SaveNeuralnetFeedbackImage",
                "value": false
            },
            {
                "key": "SaveRGBImage",
                "value": false
            },
            {
                "key": "Cast shadows",
                "value": true
            }
        ]
    },
    "colorDict": {
        "items": [
            {
                "key": "Black spot neural net color",
                "value": {
                    "r": 1.0,
                    "g": 1.0,
                    "b": 1.0,
                    "a": 1.0
                }
            },
            {
                "key": "Scratch_Color",
                "value": {
                    "r": 1.0,
                    "g": 1.0,
                    "b": 1.0,
```


SaveData - Kladblok

Bestand Bewerken Opmaak Beeld Help


```
    "floatDict": {
        "items": [
            {
                "key": "Pear spawn rate",
                "value": 0.0
            },
            {
                "key": "Pear_Dots_Amount",
                "value": 0.0
            },
            {
                "key": "Camera focal length",
                "value": 0.0
            },
            {
                "key": "Pear_Dots_Scale",
                "value": 0.0
            },
            {
                "key": "Top_Brown_Density",
                "value": 0.0
            },
            {
                "key": "Pear curvature variation",
                "value": 0.0
            },
            {
                "key": "Rot_Amount",
                "value": 0.0
            },
            {
                "key": "Light intensity",
                "value": 0.0
            },
            {
                "key": "Black_Spot_Amount",
                "value": 0.0
            },
            {
                "key": "Pear curvature X",
                "value": 0.0
            },
```


Figure 33 Part of the settings' JSON file

Connecting the settings to the preview:

With the settings correctly saving and loading, I started to connect the settings to the preview. After learning how to set Substance graph properties through code, I connected the sliders and colour pickers to the graph. When changing the sliders however, the regenerating of the graph outputs took some performance, so to make the UI feel smoother I determined at which points the graph should update. For example, the graph only gets updated when releasing the slider, or when the user is finished with typing a value in an input field.

Testing the UI:

After implementing the settings, I conducted a user test with my colleagues. I decided to give the users a small list of tasks which to follow. While doing the tasks, I made observations, and after the tasks were completed, an interview was conducted to get deeper insights into the pitfalls of the settings menu. The tasks and conclusions can be found in appendix 7.

Changes to the UI:

One of the things that became apparent when looking at the results was that the collapsible setting blocks were not easily recognized as such. To improve this, I made them open by defaults as opposed to closed. Another change I made was changing the save and load settings button. In the test I placed the buttons in the top left and top right of the preview panel, but in the interview several users indicated that they would prefer it if the buttons could be together, thus I grouped the buttons and placed them in the middle right of the preview panel. In figure 34 we can see the final UI.

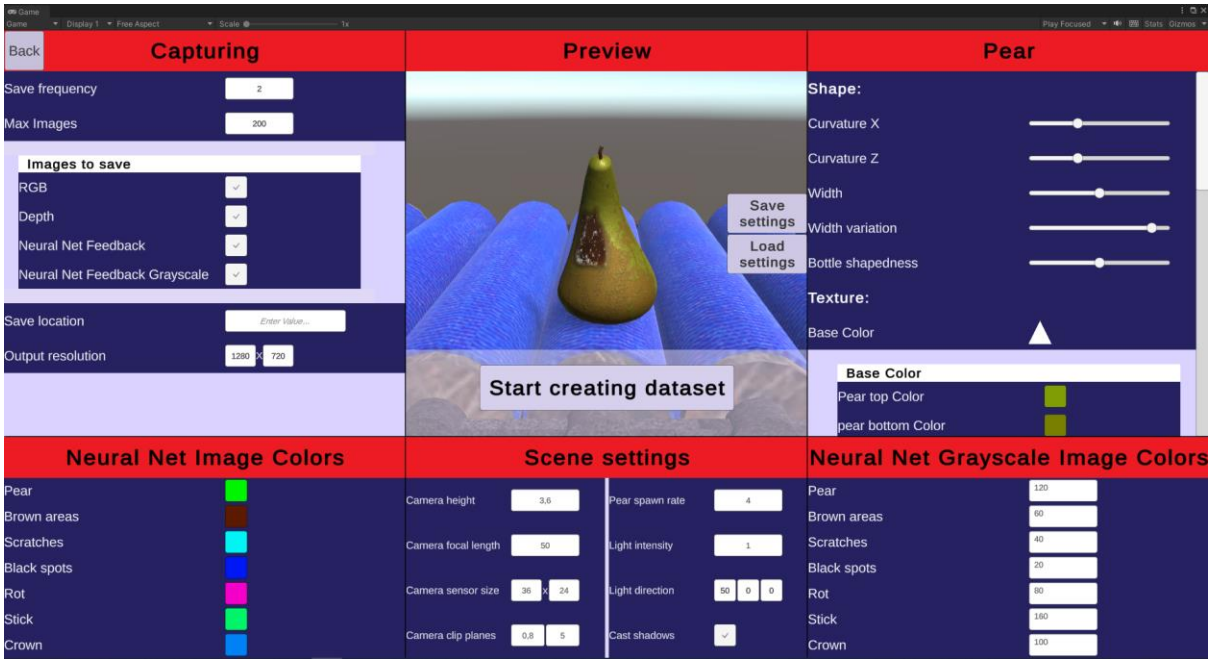


Figure 34 Final UI.

Making automated annotations:

Converting images to point clouds:

Until this point in the project, I made RGB, depth and semantic segmentation images. The neural network does however not expect an image with distinct colours but instead it wants a COCO file with coordinates that make up a polygon, and an index indicating what the polygon is. This JSON file is normally manually created by annotating images in a tool called LabelMe (see Appendix 8 for an overview of LabelMe) and converting the LabelMe JSON file to COCO with a too called LabelMe2COCO. To decrease the manual labour required to use my tool, I made an algorithm that translates my images into LabelMe's JSON format (see figure 35).


```
{
  "label": "Russeting",
  "points": [
    [
      234.5,
      447.5
    ],
    [
      233.5,
      447.5
    ],
    [
      233.5,
      446.5
    ],
    [
      234.5,
      446.5
    ]
  ],
  "groupId": 0,
  "description": "",
  "shape_type": "polygon",
  "flags": []
},
{
  "label": "Flesh",
  "points": [
    [
      235.5,
      394.5
    ],
```


Figure 35 JSON file showing several points that make up an annotation.

After much testing, I settled on an implementation that made use of 2D bit masking. This is a technique frequently used in tile-based games to determine what terrain tile should be placed based on the surrounding tiles (Bone, 2016). In my algorithm, I loop over each pixel in the feedback image and compare the pixel to the colours the user set in the settings. If the colour matches, the system creates a group of all connecting pixels with the same colour. In the end we have a list with all groups or "blobs" in the image. For each blob we use the bit masking technique to get all the points that make up that blob. After looping over each blob and writing the points to a JSON file, we get data in the format the neural network expects.

Unexpected behaviour:

When looking at the generated output in LabelMe, I saw that most points were correctly annotated however, there were also issues like points missing, sets self-intersecting, sets missing from the file or pears having the completely wrong annotation. These errors can be seen in appendix 9. After several days of looking into the issues, I discovered that the issues were caused by a small error in my code. Instead of removing the first and last item from a list, and then adding a new item, I added the new item and then removed the first and last item. This caused the newly inserted item to be removed again, and one of the old items to remain. By swapping the order all issues went away except for an issue related to holes in the blobs.

Connecting holes:

With the algorithm I implemented, blobs can have multiple sets of points if the blob has one or more holes in it. The JSON files made by LabelMe do not support polygons with holes, so my supervisor came with a solution where holes connect to the outer contour through a connection with a width of zero. This solved the initial issue, but there were instances where this connection would go through other holes or outside of the pear if the pear is very concave (figure 36) causing the generated JSON file to not be readable by the neural network.

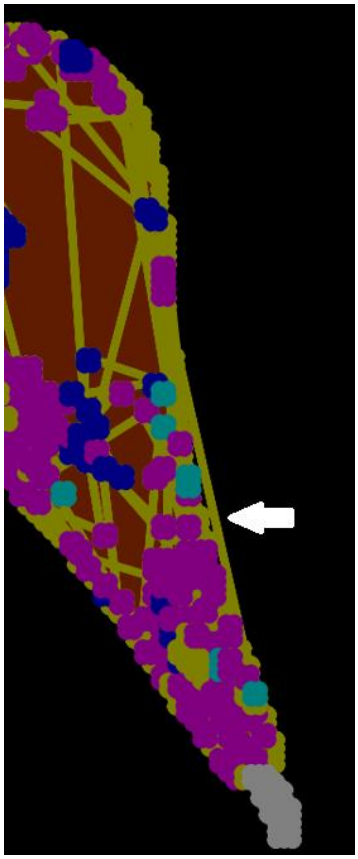


Figure 36 Connection with a width of zero (indicated with white arrow) going outside of the pear.

After looking at the problem and the sets generated points, I concluded that every blob has one outer set with zero or more inner sets. The fact that there is one set which surrounds all other sets is important as this allows the algorithm to connect all sets to each other with a zero width line by getting the right most point of the set and keep going to the right until it hits another set, or in case of the outer set doesn't hit anything. This implementation ensures that all the sets of points are connected and there is no overlap of zero width lines with holes. In figure 37 we can see the result of my algorithm when imported into LabelMe.

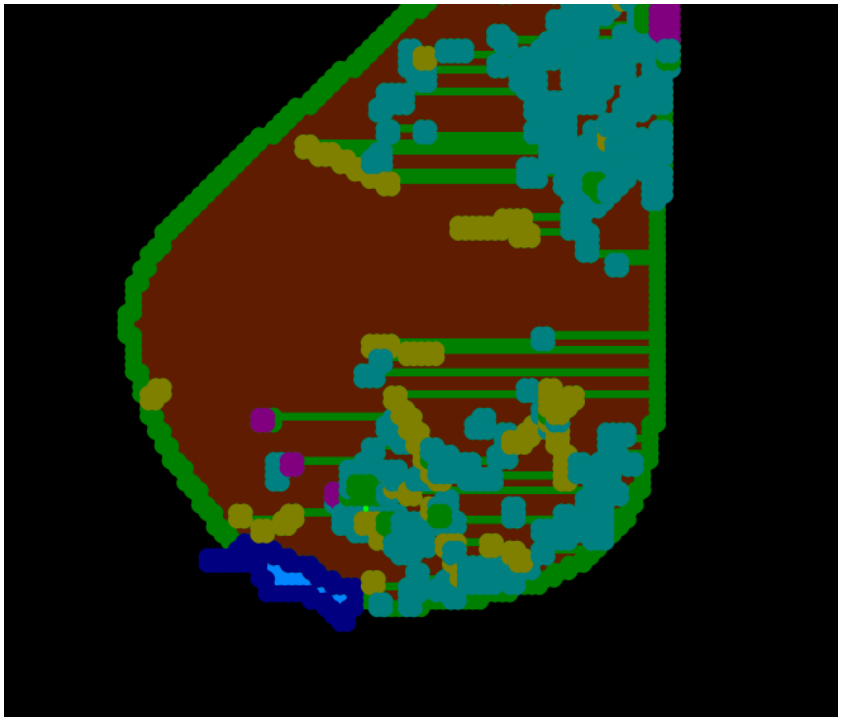


Figure 37 Correctly generated annotation

Increasing performance:

Since the annotation algorithm makes several calculations for each pixel the algorithm was quite slow. I made optimizations to the algorithm which improved the performance. The change with the biggest impact was converting the datatypes in my code from Lists to hash sets. hash sets place items in an array of lists based on the hashed item. This has the benefit that iterating over them is O(1) efficiency instead of O(n), making it much faster when looking for objects (Coders Campus, 2015).

Testing the implementation:

After having implemented the automated annotation, I made a dataset of 200 generated images with annotations in the LabelMe format and converted it into the COCO format which is a format that can be used for training the actual neural network. I handed the images and COCO file to a colleague to train a neural network. The file did however not get accepted due to hard to trace errors. After trying several potential solutions, the neural network still did not accept the data. With me being confident in my code, and with the potential issues ruled out, my supervisor suggested trying the dataset with a different neural network.

In my search for a different network, I came across MMDetection. This tool looked promising as it included several popular neural networks, it accepted the COCO format, documentation was good, and it had windows support. After going through their tutorials, I managed to set up the network and execute an example which resulted in me being detected as a human which can be seen in

figure 38. I managed to load my dataset and I successfully trained the network meaning that my automated annotation was working.

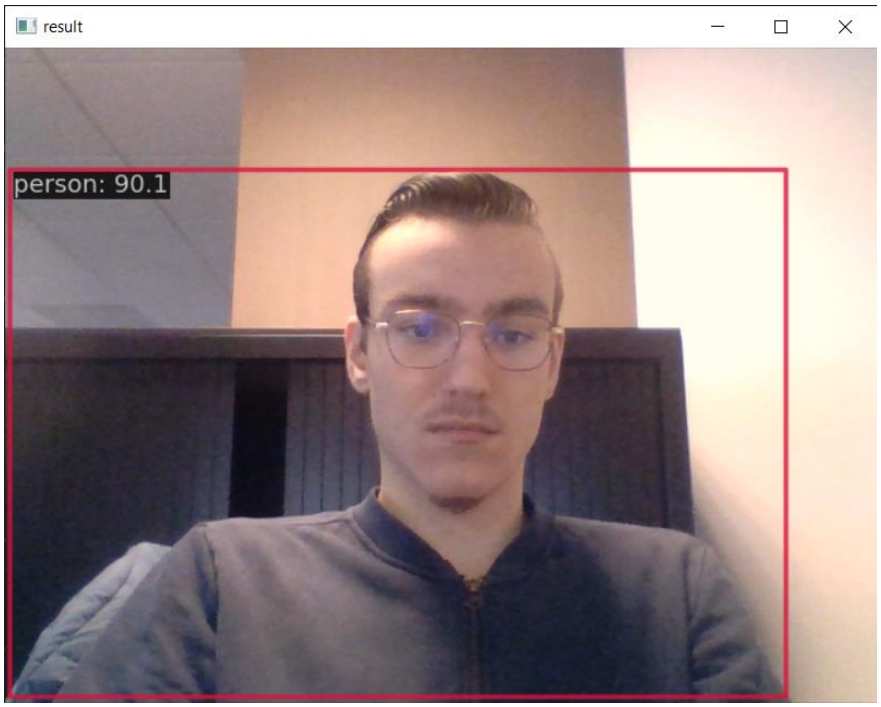


Figure 38 Tutorial neural network correctly identifying a person.

I also evaluated the result after training the network, the results of which can be seen in figure 39. The test successfully identified the pear flesh, but it did not detect all the flesh. It also did not detect the smaller features like the stick or russeting. After discussing these results with my colleague, it became clear that this was expected as I only trained for three epochs, which are similar to cycles, while a normal network is trained for 12000 epochs. I could however not train that much due to my computer not having CUDA cores. Though we will not know why the original neural network failed, my colleague and I suspect that the network cannot handle the number of points being generated.

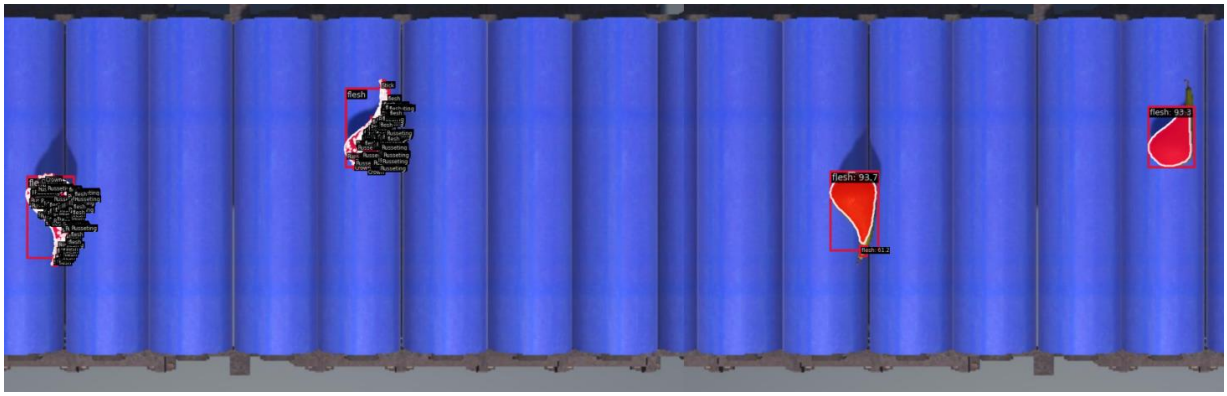


Figure 39 Neural network trained on automated annotation dataset detecting pears. The annotation can be seen on the left while the network output can be seen in the right two pears.

Implementing OpenCV annotation:

Though I have now confirmed that my annotation is working, it is still quite slow. I decided to investigate OpenCV as this is one of the most popular computer vision tools, as well as open source (6sense, 2023). Since OpenCV uses C++, while Unity uses C#, I needed a wrapper to be able to incorporate OpenCV in my application. I came across several wrappers like EmguCV, SharperCV and

opencvdotnet, but found these plugins difficult to install within Unity or having been deprecated. I ended up using OpenCVSharp as I was able to install it on both Windows and Linux without issues. The plugin's GitHub page specifically mentions that the software does not work in Unity, but by adding missing dlls to the project, and testing the functionality, I determined that this tool was suitable.

The first part of the annotation was finding the outline of the generated images. OpenCV has a useful function called find contours which does exactly that. If we use this method for every part of the pears that we want to identify, we get the contour points of the pear parts. Like before, the pear parts can have holes. Luckily OpenCV can order contours in a hierarchy, so I could connect two contours with a zero width line the same as I did in my manual implementation. This meant I had a set of points that I could use with my converter to create a LabelMe file like before.

Comparing the performance between the OpenCV solution and my manual solution, I determined that the performance is roughly 5 times faster in the OpenCV solution, so I used this implementation in the application.

There was one minor issue with OpenCV as it sets the contour points in the center of the pixel, and not on the corners. This meant that a single pixel would only have one point instead of the required four for an annotation, and that a line would consist out of two points. To fix this issue I wrote code that automatically adds points to solve this issue.

Creating documentation:

With most of the key features implemented and tested, and the deadline for the report approaching, I decided to start polishing the existing features and writing documentation. I wrote a readme file in the repository so that inexperienced users know how to install the application and work with the interface. I cleaned up my code by adding comments and formatting it in a more compact manner. For some parts of the application, I also made UML diagrams and documents which can be seen in appendix 10.

Presenting simulation at TValley Tech Conference:

Near the end of my graduation, I also had the opportunity to present my solution at Riwo's stand at the TValley Tech Conference in the Grolsch Veste (see figure 40). For this I made a demo video displaying the generated pears, the functionality of the application, and the created annotations. It also gave me the opportunity to talk with employees of other companies about how they use computer vision, simulations, and digital twins in their workflow.

Figure 40 Riwo's stand displaying my solution.

Final test:

To test if the instructions for installing my application were clear 3 colleagues also installed and tested the build like they would in a real situation. There were only a few minor issues like setting defaults not being loaded correctly, and the session data not being deleted, but these were easy to fix. Furthermore the instructions were clear and the application was easy to use.

Conclusion:

By researching several small parts throughout this project I got answers to my sub questions, and with the answers to those sub questions I implemented a working prototype thus answering the main question with: “Yes, it is possible to create RGBD images of pears in an engine, and have the generated images be connected to the ROS2 network so that testing a pear sorting robot becomes more efficient”.

Not only is it possible to create RGBD images, but it is also possible to automatically annotate them by creating a semantic segmentation using Substance Designer as the ground truth and converting the segmentation into sets of points in the LabelMe format.

With that said, it is possible that my solution might not be the most optimal given that the project was done with a limit of 20 weeks.

Recommendations:

If one were to pick up this project, I would suggest for them to read the provided documentation to navigate the project more easily. There are several components in the project that could be made better.

Currently it makes use of several cameras to capture the viewport into textures with different shaders however, cameras are slow. I suspect it would be possible to capture the scene with one camera and a more advanced shader which would result in more frames.

Another part that is quite slow is the package I used to generate the colliders. With a better system to convert mesh colliders into several cube colliders, the framerate might improve. Unity’s mesh collider works with rigid bodies in an old Unity version, so one might go back to this version however, several core packages I used do not support this old Unity version.

A pear is not a complex shape. If one were to use a similar simulation for objects that should be both procedurally generated as well as complex, it might be better to make the 3D models in a procedural modelling program like Houdini, though Houdini does not allow procedural generation in builds due to licensing issues.

Though the OpenCV annotation method correctly annotates the images, it could be more detailed as the annotation uses the center of pixels, and not the borders of pixels. If one would like to increase the accuracy by using pixel corners, they would need to either improve my manual annotation, or find a different tool to help create annotations as OpenCV does not support pixel border contours.

Perhaps the biggest point of improvement that could be done is in the fact that the annotation does not work with Riwo’s software. Further investigation is needed into the source of this problem as the created annotation works with different neural networks.

Reflection:

Looking back on the project, there are several things that went well, while a few things could have gone better. One of the things went well was the division and planning of the project. When I joined the company, the company supervisor advised me to deliver something small every sprint instead of a big broken feature. Throughout this project, I mostly finished my tasks for that sprint, and planned my tasks accordingly, the only component that was not finished at the end of a sprint was the annotation, but this was partially due to it being added later in the sprint.

I think the communication with the company supervisor also went well. Every week we had a meeting discussing the project and the progress made on it that week, and in case I was stuck on anything, if I needed assistance with anything.

One of the things that I think could have been done a little more extensively was validating the created images with the network. The reason I did not do this much was due to the network not being fully developed yet. I could have done some testing in a different way, but with the fact that the users can define the shape of the pear however they like, I determined that this was redundant.

Throughout the project the team also held sprint meetings in which we show our progress to other team members through a small presentation or video as well as a sprint report. Though my presentations showed the progress well, I did not spend much time on making them look all that pretty. This however did not seem to be an issue as I never heard any remarks about it.

Through user testing I concluded that my UI was functional and not bad looking however, If I had spent more time on creating custom assets for my UI, I think I could have made it look more pretty.

Reference list:

6sense. (2023, May 24). *OpenCV - Market Share, Competitor Insights in Data Science And Machine Learning*. https://6sense.com/tech/data-science-machine-learning/opencv-market-share

Aderinola, B. (2022, November 18). *What is Gazebo Simulation - The Construct*. The Construct. https://www.theconstructsim.com/ros-5-mins-028-gazebo-simulation/

Bedell, C. (2021). big-endian and little-endian. *Networking*. https://www.techtarget.com/searchnetworking/definition/big-endian-and-little-endian

Bone, S. (2016). How to Use Tile Bitmasking to Auto-Tile Your Level Layouts. *Game Development Envato Tuts+*. https://gamedevelopment.tutsplus.com/how-to-use-tile-bitmasking-to-auto-tile-your-level-layouts--cms-25673t

Coders Campus. (2015, October 6). *Java HashSet Tutorial* [Video]. YouTube. https://www.youtube.com/watch?v=WPcKwA5WF7s

Druguet, F., Drettakis, G., Girardeau-Montaut, D., Martinez, J. L., & Schmitt, F. (2004, July). *Figure 7: Unfolding a cylinder to a plane.* ResearchGate. https://www.researchgate.net/figure/Unfolding-a-cylinder-to-a-plane_fig12_220955363

Hahmann, S., & Bonneau, G. P. (2008). *Polynomial surfaces interpolating arbitrary triangulations*. University of Grenoble. https://hal.science/hal-00319652/document

Jayelinda. (n.d.). *Modelling by numbers: Part One A | Jayelinda's Web*. http://jayelinda.com/modelling-by-numbers-part-1a/

Kalathil, J. (2018, December 22). Introduction To Replacement Shaders & Shader Keywords. *Bitshift Programmer*. https://www.bitshiftprogrammer.com/2018/12/introduction-to-replacement-shaders-and-shader-keywords.html

Sanukin. (n.d.). *UniColliderInterpolator*. GitHub. https://github.com/sanukin39/UniColliderInterpolator

Understanding nodes — ROS 2 Documentation: Foxy documentation. (n.d.). Retrieved June 12, 2023, from https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html

Unity Technologies. (n.d.-a). *Unity - Manual: Mesh Collider component reference*. https://docs.unity3d.com/Manual/class-MeshCollider.html

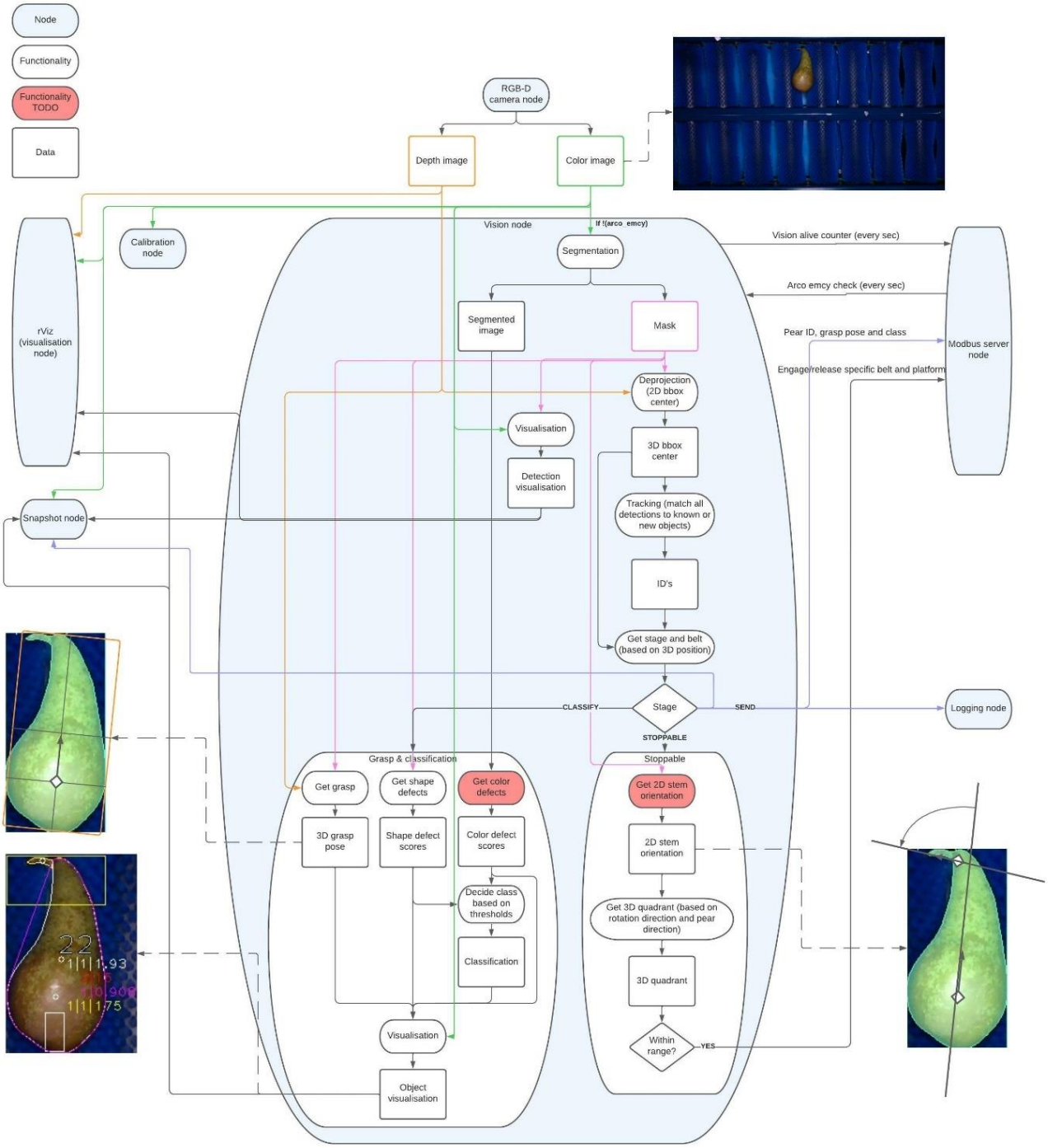
Unity Technologies. (n.d.-b). *Unity - Manual: ShaderLab: Culling & Depth Testing*. https://docs.unity3d.com/2017.2/Documentation/Manual/SL-CullAndDepth.html

Unity Technologies. (n.d.-c). *Unity - Scripting API: MinMaxCurve*. https://docs.unity3d.com/ScriptReference/ParticleSystem.MinMaxCurve.html

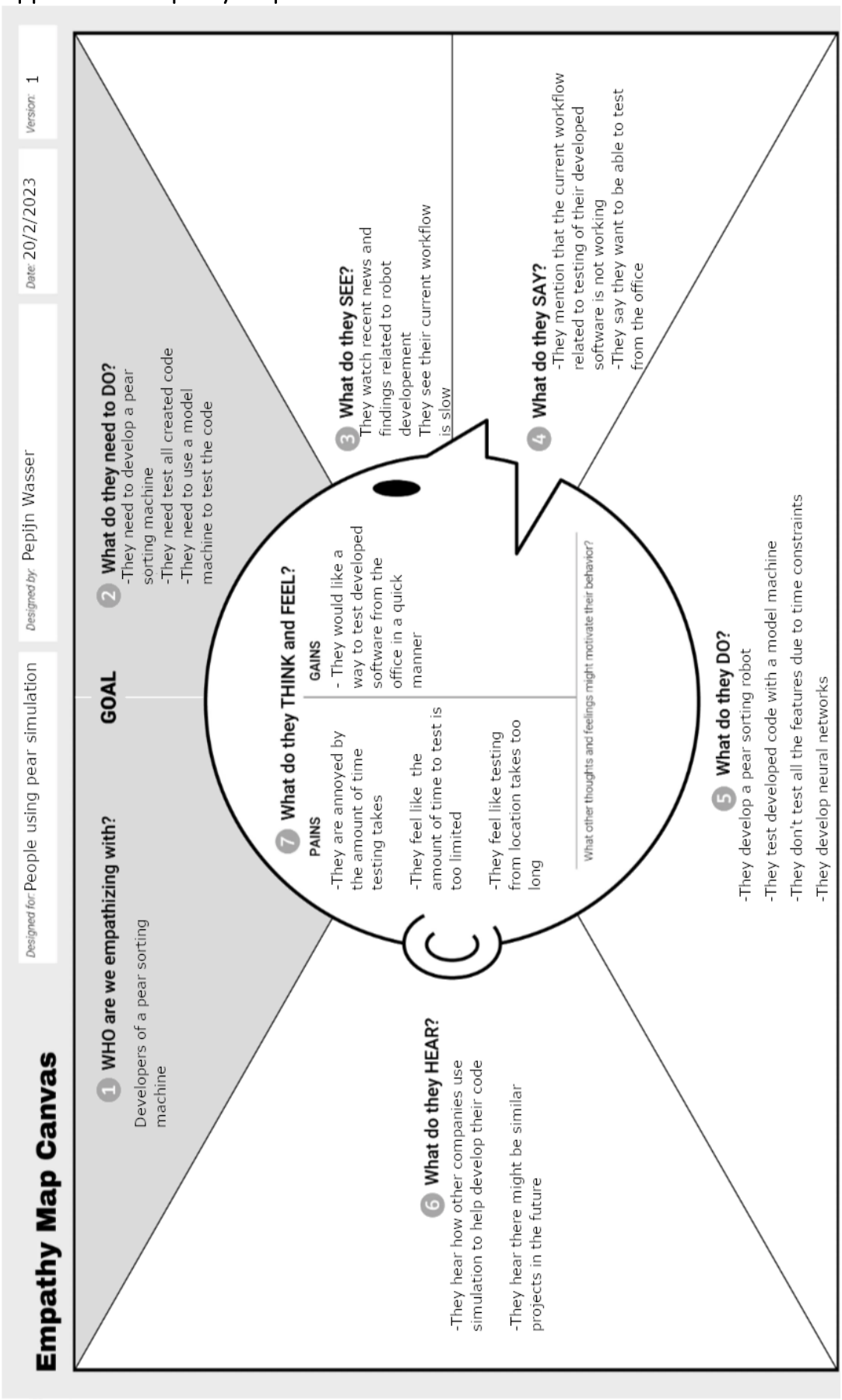
Wkentaro. (n.d.). *LabelMe*. GitHub. https://github.com/wkentaro/labelme

Appendices:

Appendix 1: Pear packaging component overview



Appendix 2: Empathy map



Appendix 3: Document describing pear quality.

Context:
For my graduation project I am generating pears to simulate camera data from a pear packaging and sorting machine, but to generate good and bad pears, I need to know what a good pears and bad pears are. This document goes into detail on how to determine the quality of pears.

Pear quality:
To determine what makes for a good or bad pear, the customer of RIWO came up with several guidelines. They divide pears based on a variety of aspects like: shape, curvature, size, colour, the stick, internal injuries, external injuries and roughness. With these aspects in mind, they dive the pears in three quality groups (Q1, Q2, Q3).

For reference material, the customer send hundreds of images of pears sorted with their respective quality group and several tens of pears for each category.

Q1:
Pears with Q1 quality are the healthiest and most beautiful pears, hence why they are sold at supermarkets and greengrocers.

As can be seen in the image below, a Q1 pear has a nice shape with a ball-like bottom and a cone like top. The stick that sticks out of the top is long, and the pear is not injured. As the brown spots are part of the pear and they do not influence the taste of the pear, the quality is not influenced by the brown spots.

Q2:
Q2 pears are pears with minor issues. They might be slightly misshapen or they might have a unappealing colour. They are still perfectly edible thus they are used as cut pieces in fruit salads or other processed food.

Quality 2 pears can be misshapen pears or slightly bruised pears. In the images below you can see that quality 2 pears can have a lot of curvature or be bottle shaped. One can also see that some pears have slight spots that do not pierce the skin of the pear. The colour of the pear can also be slightly more brown compared to Q1 pears. Sometimes the stick goes in between the rollers of the conveyor. Pears without sticks can cause rot, so when a stick is not visible, one needs to differentiate the pears based on other aspects.

Q3:
When a pear is so injured that it might cause rot, the pear belongs to quality 3. Apart from badly injured pears, wildly misshapen pears also belong to this category. Quality 3 pears are not suitable for human consumption, and will be used as animal food.

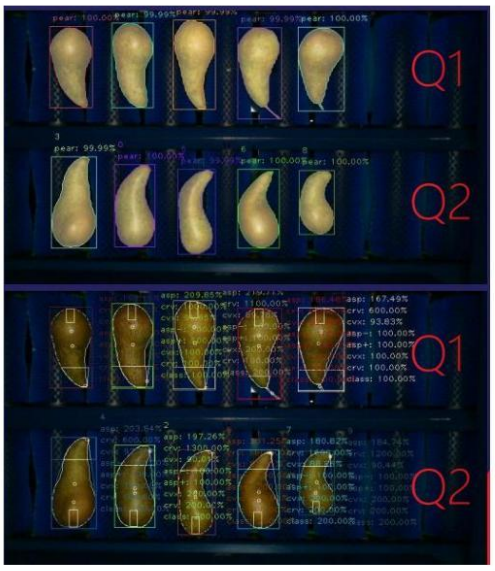
As can be seen below quality 3 pears are badly injured pears, some pears appear to be fine, but they might be missing their stick, which can result in rotting. They can also be in quality 3 due to a brown colour. Pears can be even more misshapen compared to Q2, or they can have a rough surface.

Additional points of interest:

The images you see in this document were taken on a test conveyor belt. In the real factory the pears just get out of a water bath, making them wet and adding reflections to the image.

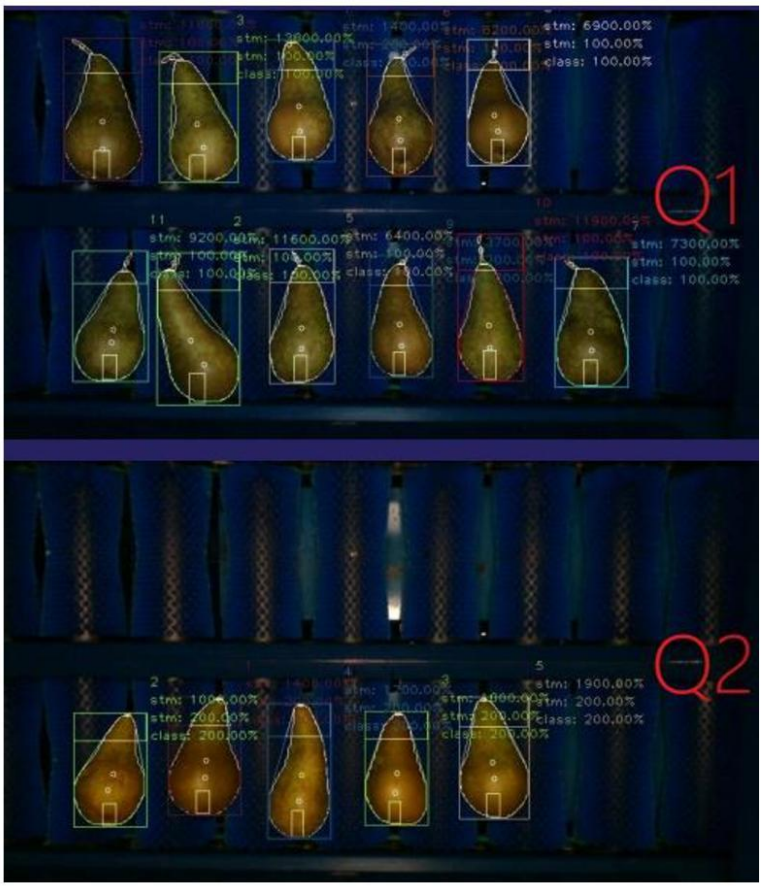
When pears move on the conveyor, some rotate themselves between the rollers. This means that the all sides of the pear will become visible to the camera. Pears with a long stick, or pears a lot of curvature do not rotate however, making it so their bottom side is never visible to the camera.

More images:

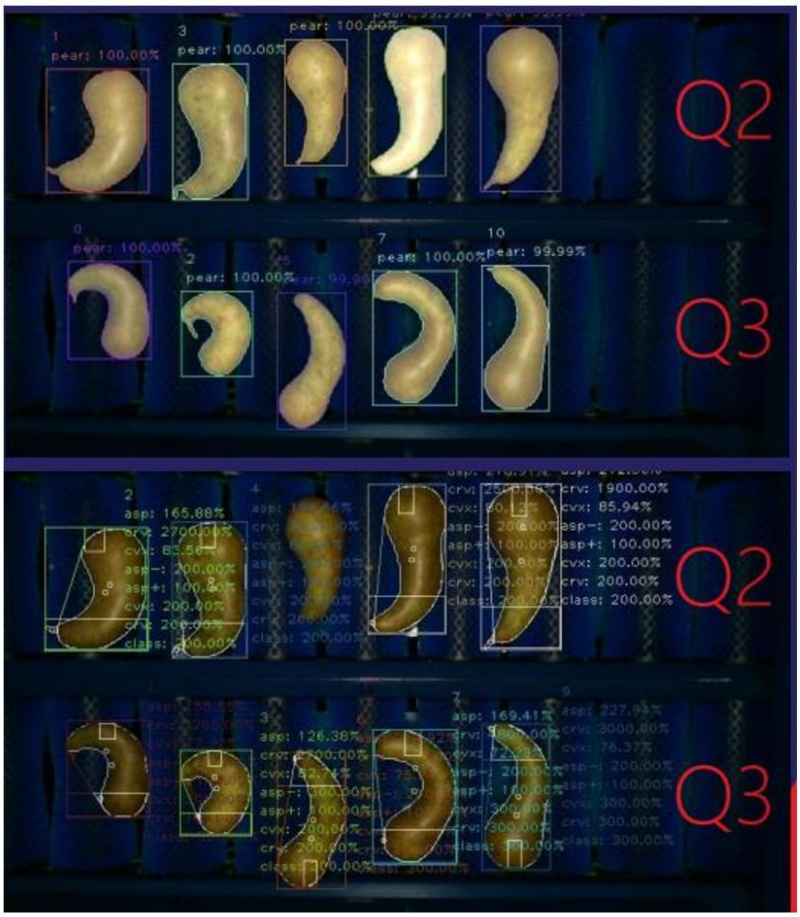


Differences between Q1 and Q2 based on curvature

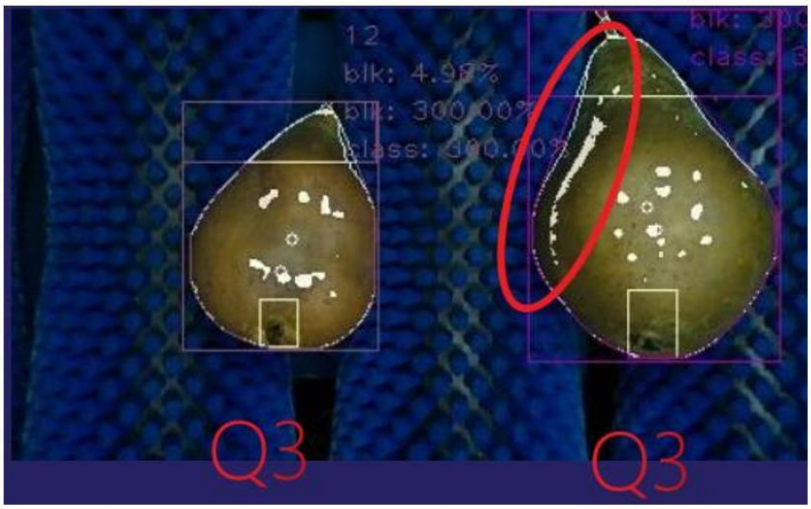
Differences Q1 and Q2 based on shape



Differences Q1 and Q2 based on stick



Differences Q2 and Q3 based on shape



Q3 pear based on roughness

Differences Q1, Q2 and Q3 based on colour

Appendix 4: SWOT analysis on possible ways to generate pears.

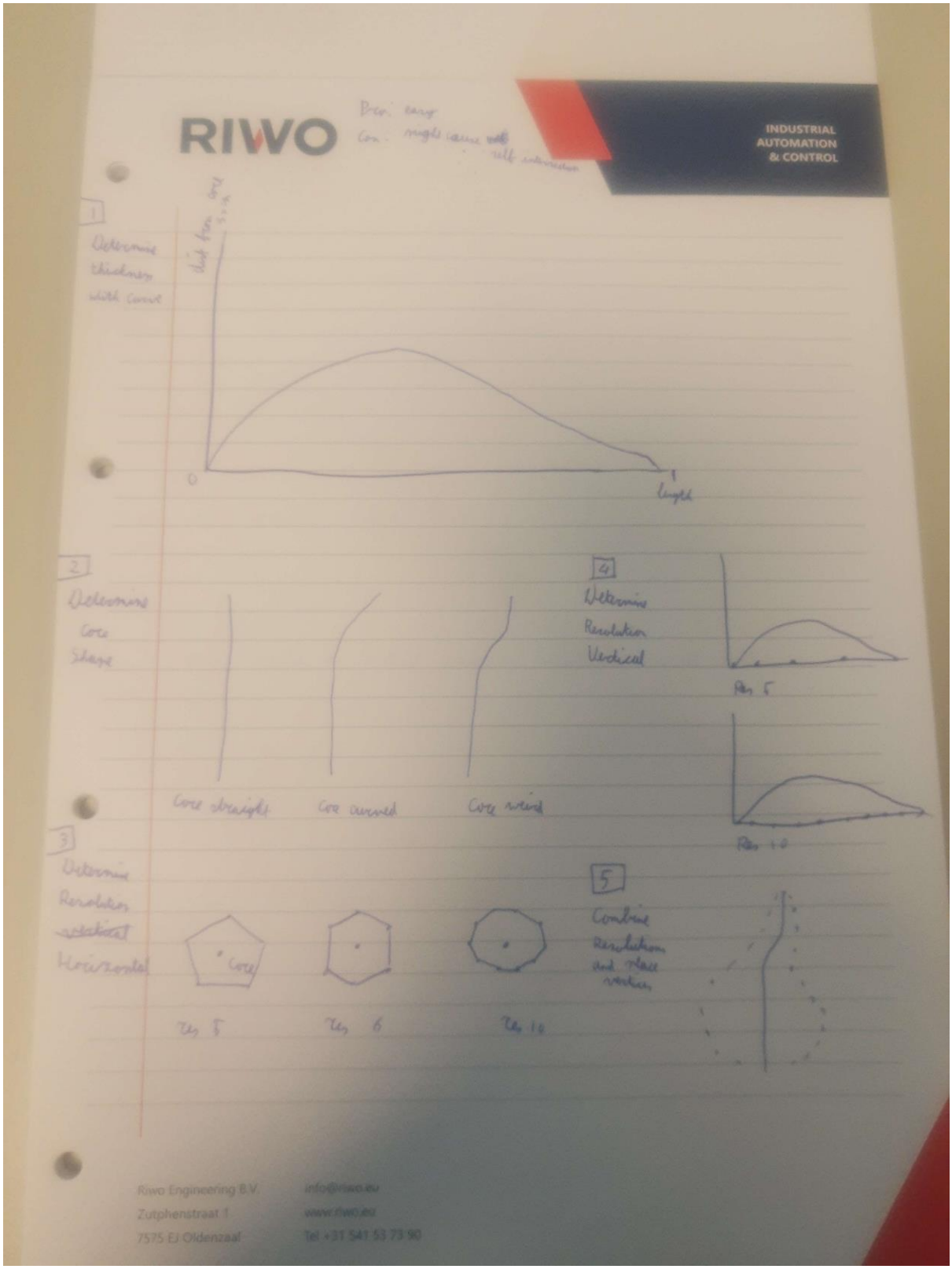
Solution:	Have program warp Images
Strength	Weakness
Easy to implement	Hard to make 3D model
	Hard to rig 3D model
	Limited user controll
Opportunity	Threat
Might work with different fruits	Might be hard to add more variations in the future

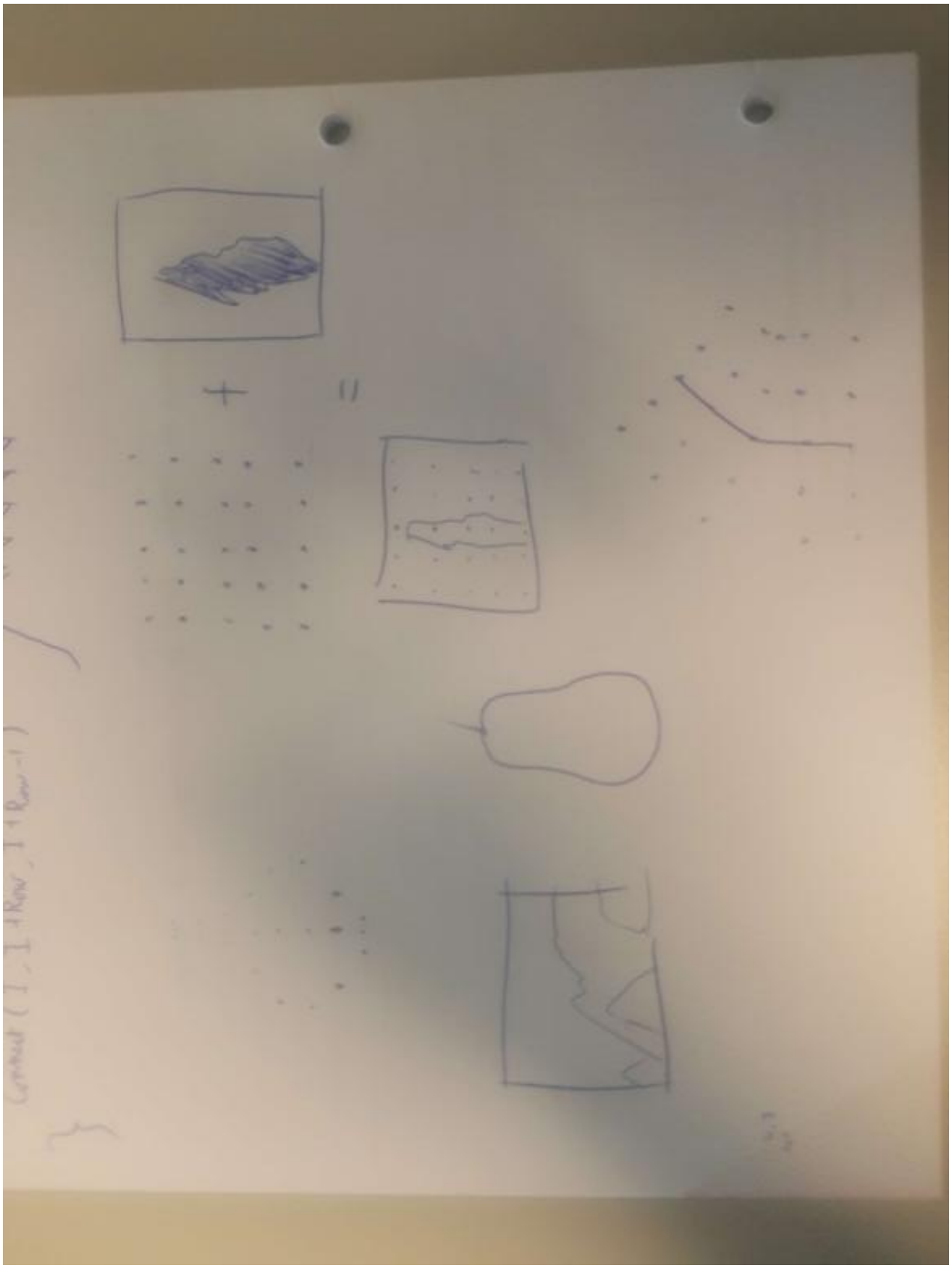
Solution:	Create 3D models and use cameras
Strength	Weakness
Full user controll	Hard to create 3D model from scratch
	Might cause self intersection in model
	UVs might be hard to set
Opportunity	Threat
Method allows the use of any shape possible	Faults with color might be hard to add due to UV maps

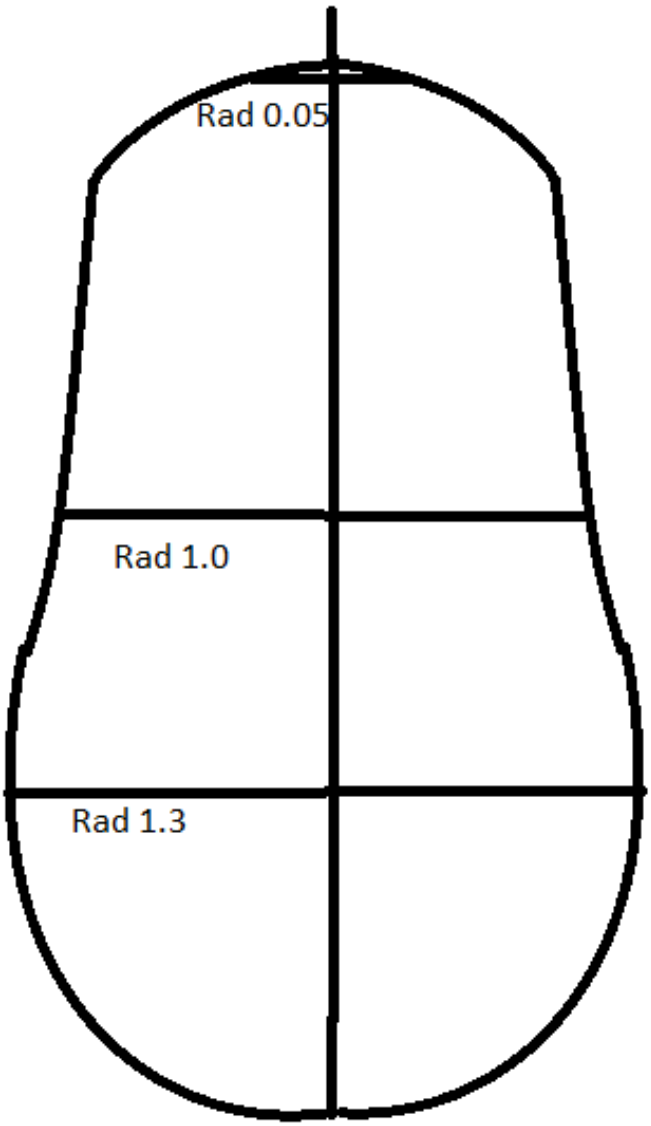
Solution:	Create procedural 3D models in Houdini
Strength	Weakness
Full user controll	Need to learn Houdini
	Houdini does not allow generation in Build
Opportunity	Threat
Houdini is powerfull and can make any shape imaginable	Houdini model generation might be too slow for runtime generation

Solution:	Train two Gan NN and have it generate based on parameters
Strength	Weakness
Creates images that are as realistic as it can get	Needs either a base dataset with annotation
	Really slow to train
	Might be hard to have it generate based on parameters
	Dubious if doable in limited time
Opportunity	Threat
This method allows for any type of image to be generated	Adding new features requires training a new model

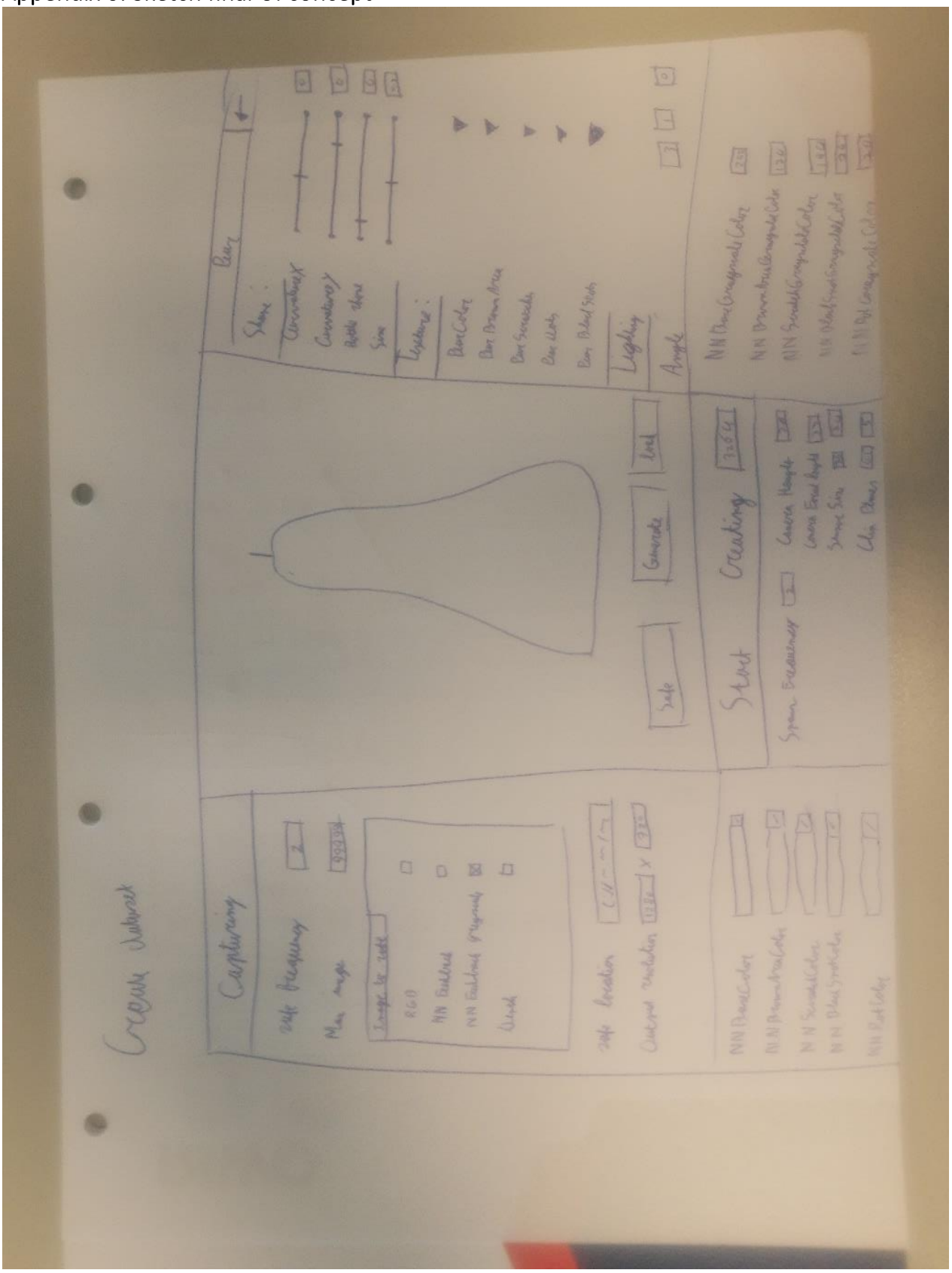
Appendix 5: Example drawings to help understand what a pear is.






Rad 0.05
Rad 1.0
Rad 1.3

Appendix 6: Sketch final UI concept



Appendix 7: UI user test results

Tasks:

1. Go to the create dataset menu
2. Set the output resolution to 1920x1080
3. Set the images to save to: RGB, Neural Net Feedback and Neural Net Feedback grayscale
4. Set the pear colour of the Neural Net Feedback image to black
5. Set the scene light intensity to 2
6. Set the pear to have no curvature
7. Set the pear to be verry bottle shaped (very straight)
8. Set the scratch colour to bright blue
9. Set the scratch size to 0,10 x 2
10. Save the settings
11. Change any setting
12. Load the settings
13. Make a screenshot (windows + shift + s)
14. paste the screenshot in paint (ctrl + v)
15. Open the save data file (already opened in the task bar)
16. Disable cast shadows in the save data file
17. In the application load the settings
18. go back to the main menu (where completed step 1)

Method:

After completing the tasks the participants were asked to give their thoughts. This was followed by asking questions regarding observations during the completion of the tasks and previous answers. Questions were also asked regarding UI elements, positioning and functionality. Lastly the interview ended by asking the participant for any more feedback that might have come up during the interview.

Participant 1:

Observations:

1. Was confusing neural net grayscale colours for neural net colours
2. Took a bit of time to find the dropdowns

Interview results:

1. Would like the load settings to be underneath the save settings
2. Setting the pear scratch size would not allow decimal numbers
3. For the amount of settings the user interface was clear
4. Saw that the settings were divided into blocks and roughly saw the theme of 3 blocks

Participant 2:

Observations:

1. Participant changed the neural net scratch colour instead of the pear scratch colour.
2. Took some time for the participant to see the dropdowns
3. Did not notice that changing the external files and loading them had any influence on the settings

Interview results:

1. Participant expected to receive a popup with the save location when saving the options
2. Because of point 1, the participant was confused what the output location field in the capture block was for.
3. User was confused about the naming difference between neural network feedback image in the task list and neural network image setting block in the user interface.
4. The load settings and save settings button being apart was not optimal
5. A popup saying settings have been saved/loaded would have been nice
6. Setting blocks appearing when certain boxes were ticked was confusing as you were clicking in the middle of the screen, while things appeared/disappeared in the bottom of the screen
7. Participant was able to recall all setting blocks and their functionality

Participant 3:

Observations:

1. Participant changed the neural net scratch colour instead of the pear scratch colour.
2. Participant was confused by the ZCurve not working

Interview results:

1. Participant did not like the colour palette
2. Participant would make all the setting blocks smaller and organize the window differently
3. Participant did not have any issue with the positioning of the back/save/load button
4. Participant felt that the UI looked blocky
5. Participant did not find the dropdown arrows clear
6. Neural net setting blocks disappearing was not distracting
7. Participant would be able to quickly interact with the interface after a small overview
8. Participant would like there to be some text giving meaning to the sides of the slider
9. Participant did not realize that the preview is a preview.

General observations:

1. Settings do not load for the second time when going back to the menu, and then going back to the settings again.
2. Neural net image setting blocks do not automatically open when the menu is loaded
3. The loading of the settings menu takes too long causing confusion if the menu is working
4. ZCurve not working

Key takeaways:

1. When the user presses a button, there should be feedback (When loading the scene, saving settings and loading settings)
2. The preview should have a title
3. It needs to be clearer that the dropdowns are actually dropdowns
4. If possible it would be nice to have a UI with more rounded edges
5. Save and load settings should be close to each other
6. Perhaps make the headers bigger to make the user more aware of the different setting categories

Participant 4:

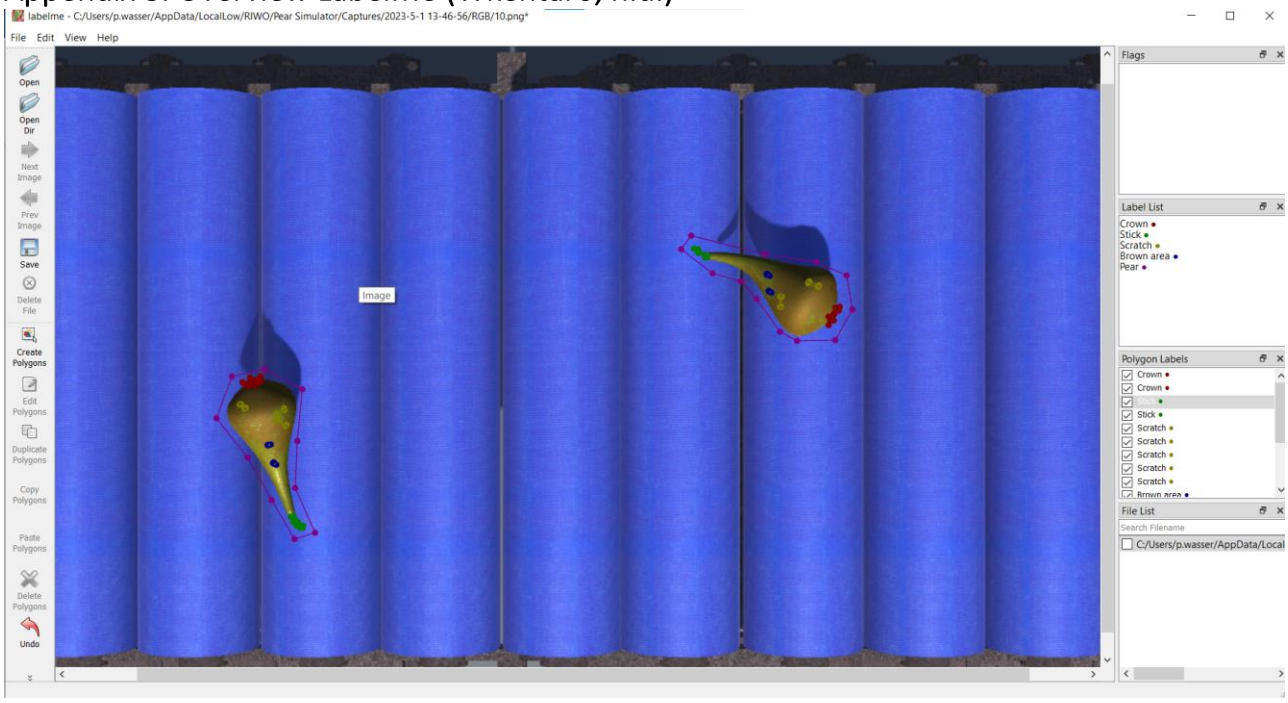
Observations:

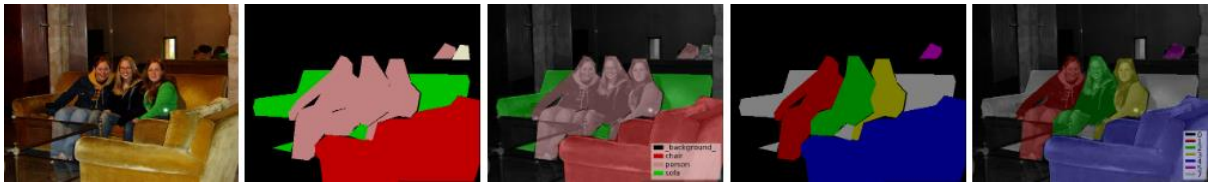
1. Participant confused the neural net image colours with the pear colours
2. Participant tried to click on the colours in the colour picker panel to move to that hue instead of moving the slider.

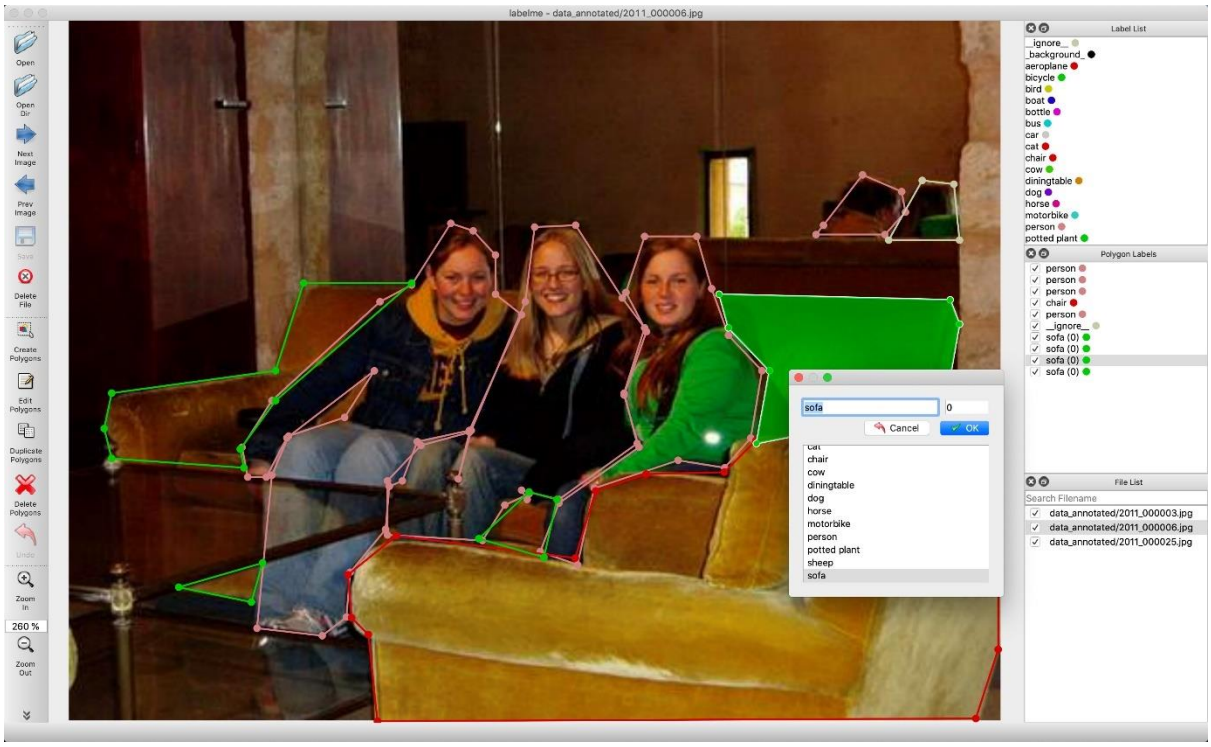
Interview results:

1. Participant thought the UI was clean
2. Participant liked that the pear settings which influenced the preview were next to the preview
3. Participant recommended to switch around the neural net image colours with the scene settings
4. The location of the save/load and back buttons were clear
5. Participant has never worked with JSON files
6. Participant saw that the settings were divided into blocks

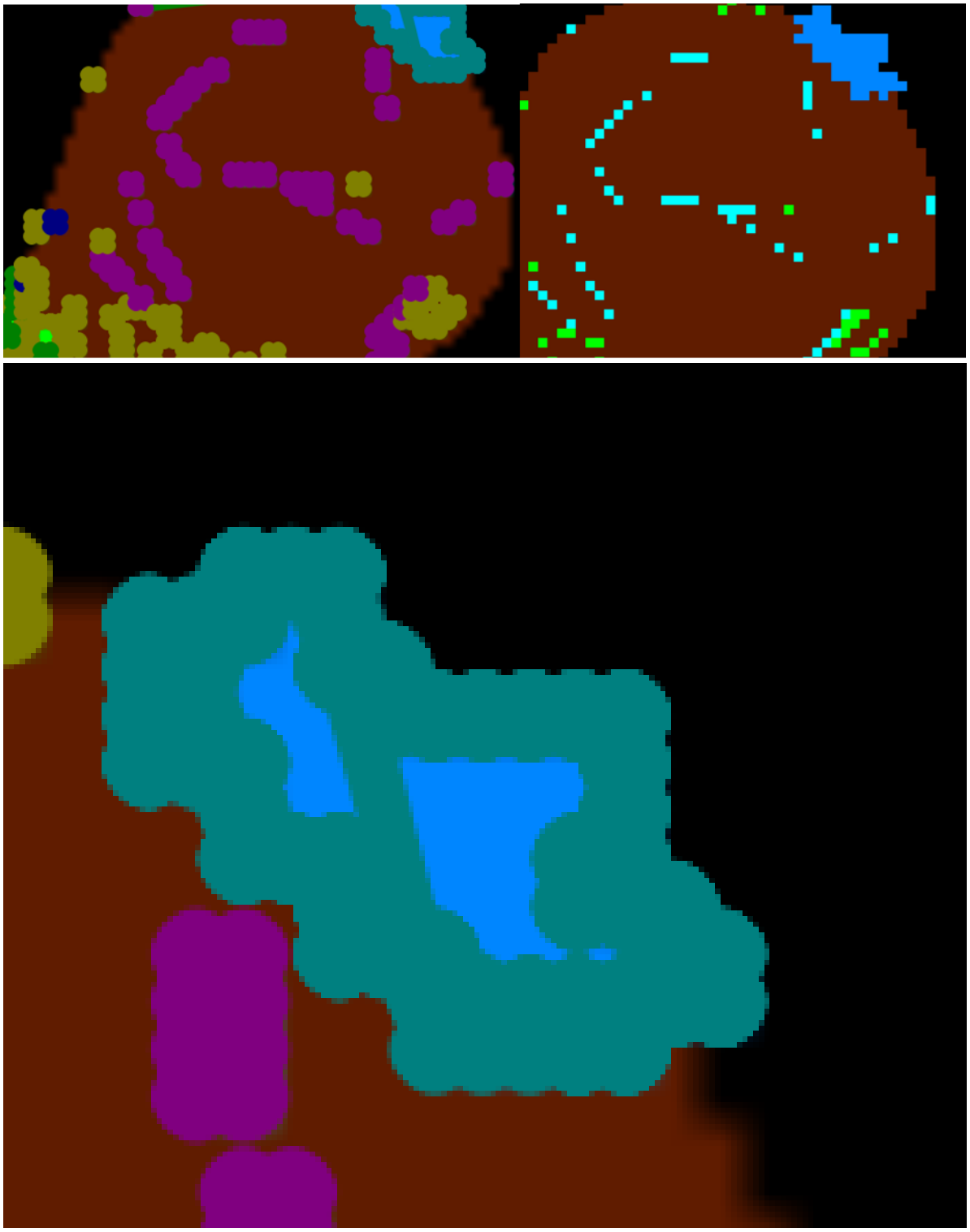
Appendix 8: Overview LabelMe (Wkentaro, n.d.)

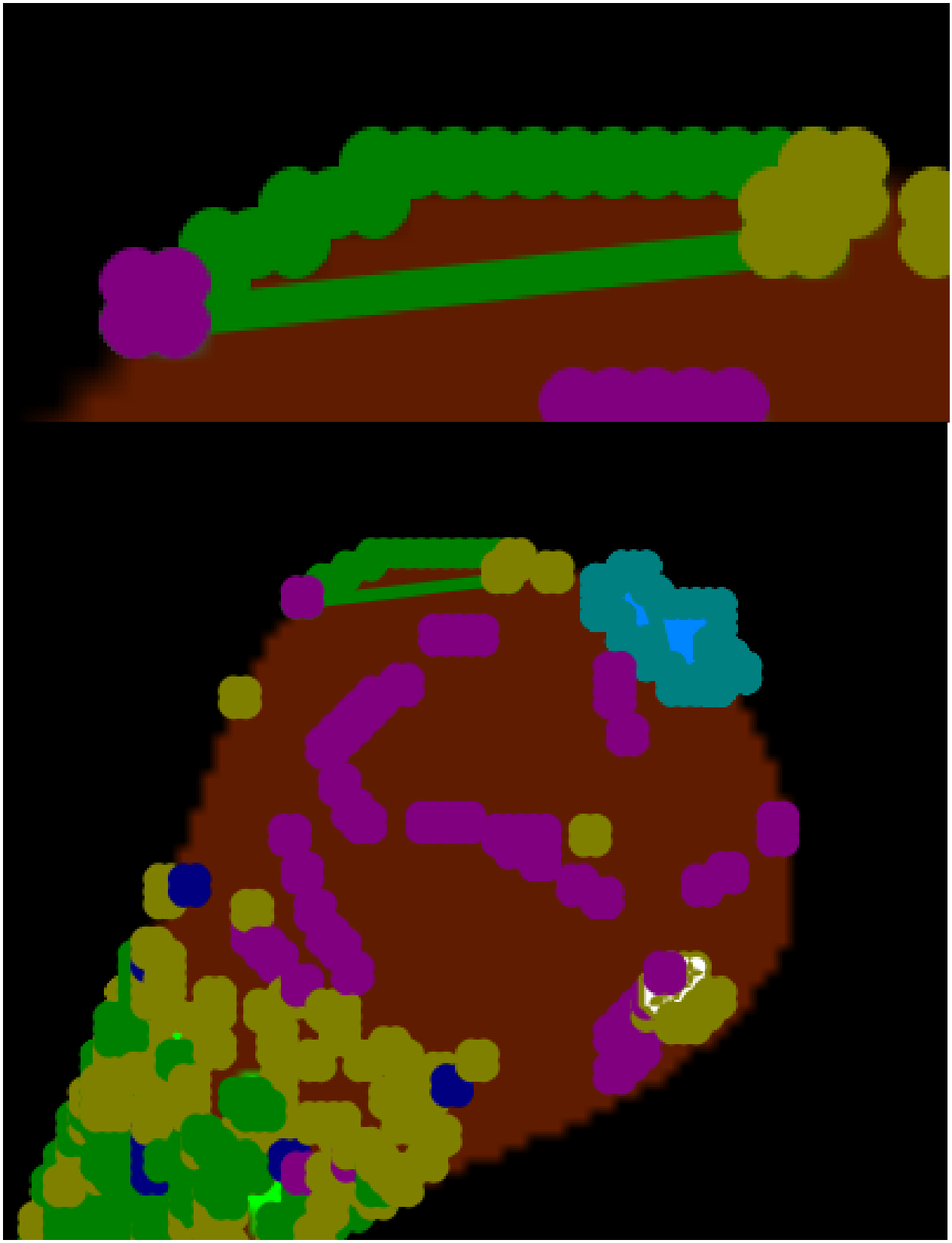


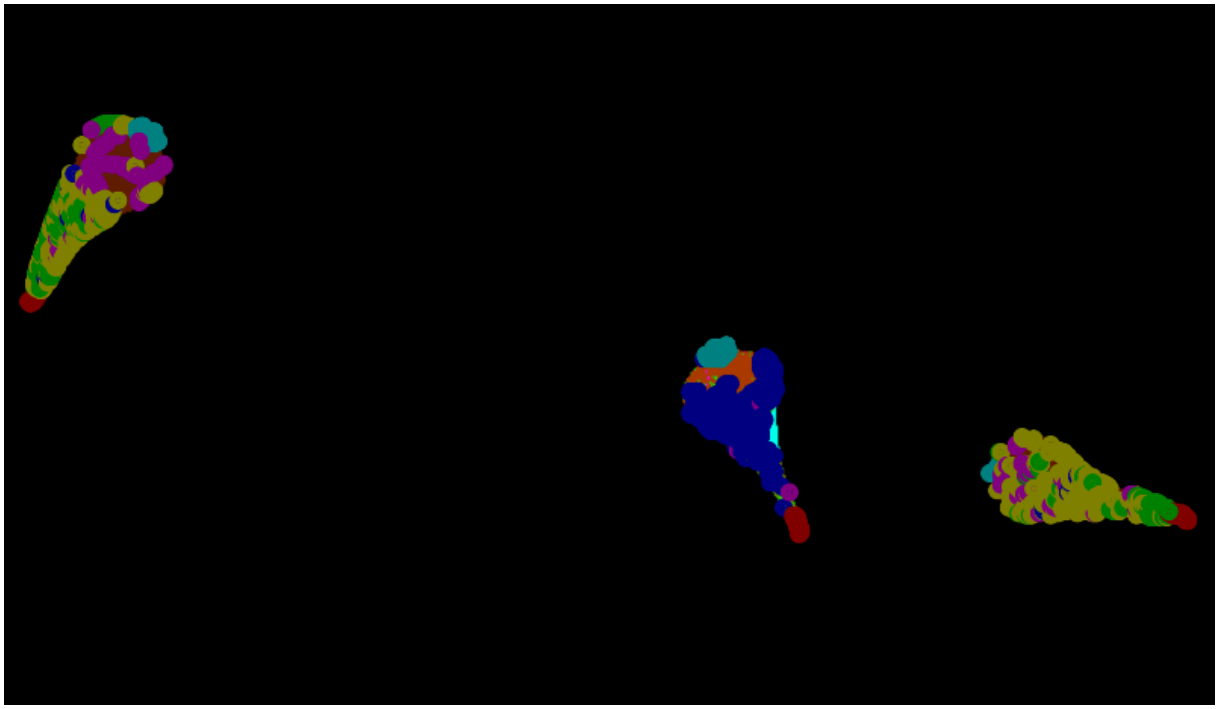




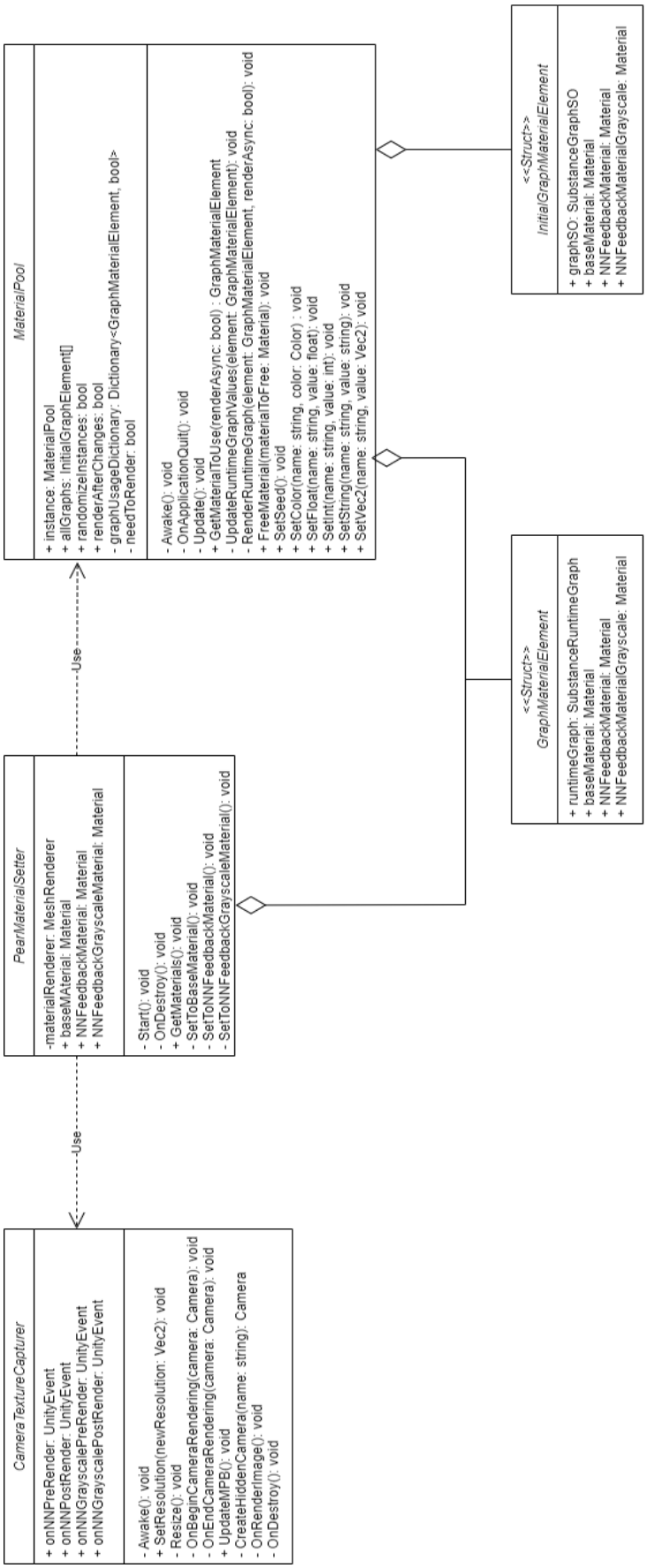
Appendix 9 Resulting annotation with errors from first iteration of automated annotation

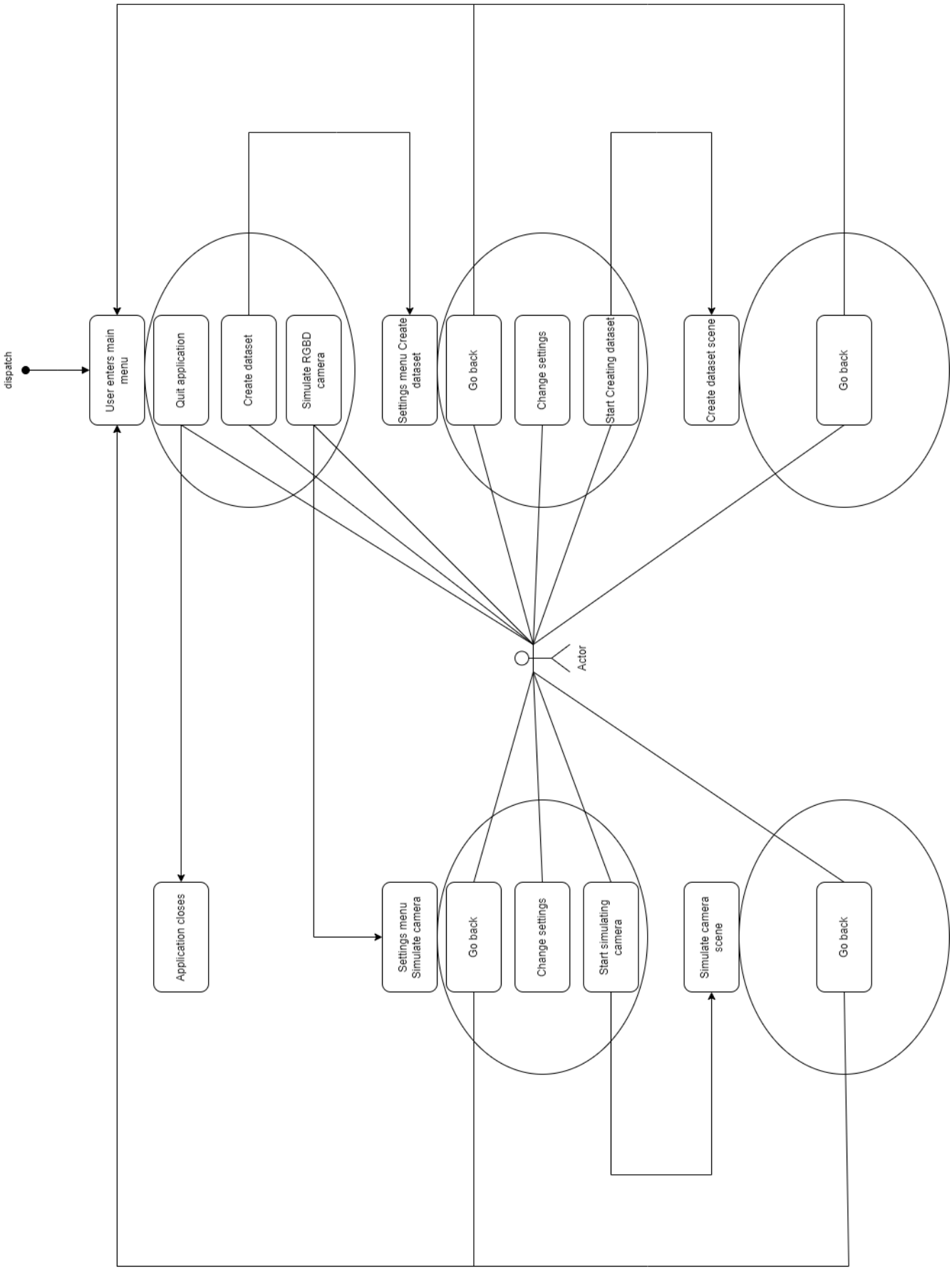


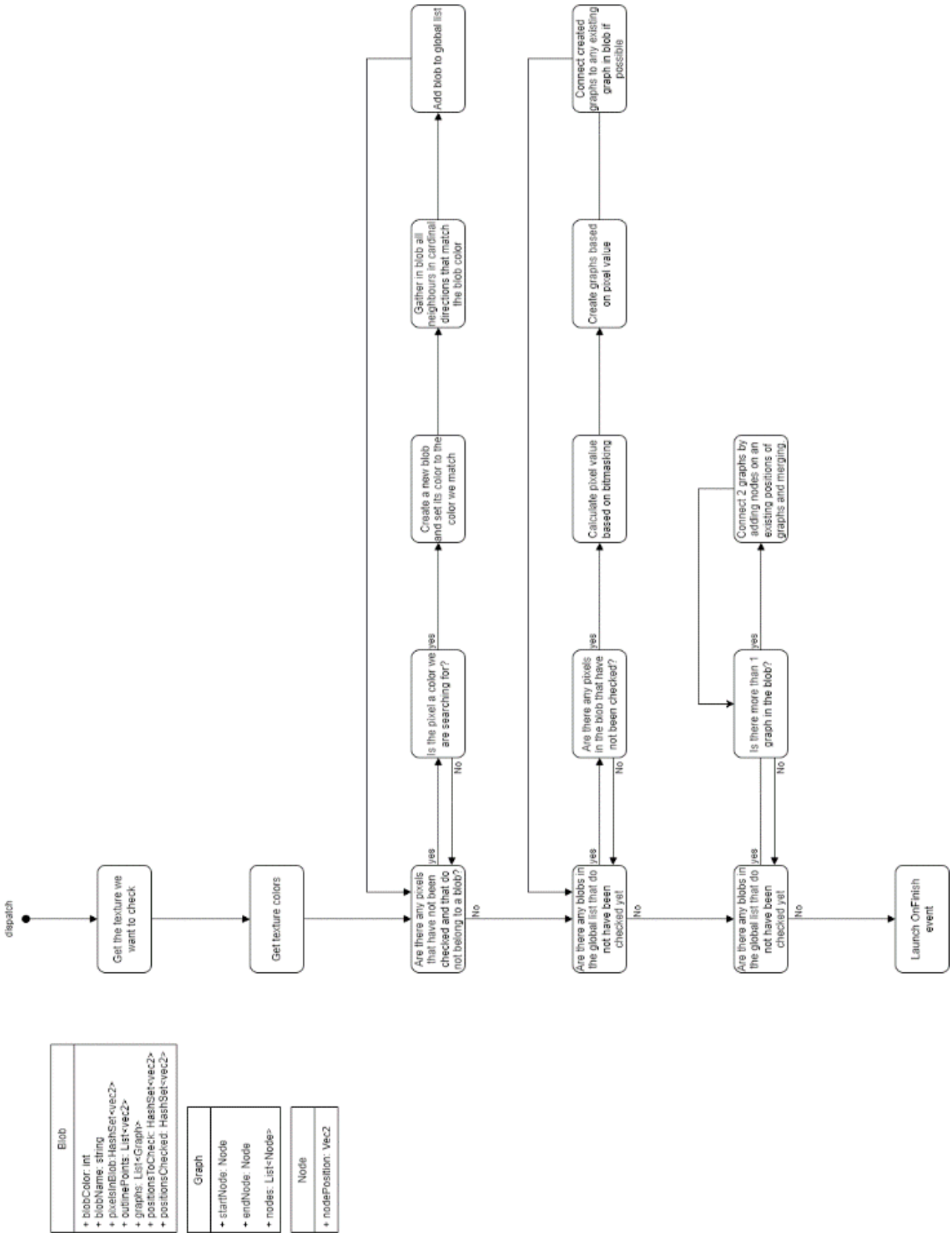




Appendix 10: Several UML diagrams and documents




dispatch
User enters main menu
Quit application
Create dataset
Simulate RGBD camera
Settings menu Create dataset
Go back
Change settings
Start Creating dataset
Create dataset scene
Go back
Actor
Application closes
Settings menu Simulate camera
Go back
Change settings
Start simulating camera
Simulate camera scene
Go back


dispatch
Get the texture we want to check
Get texture colors
Are there any pixels that have not been checked and that do not belong to a blob?
yes
No
Is the pixel a color we are searching for?
yes
No
Create a new blob and set its color to the color we match
Gather in blob all neighbours in cardinal directions that match the blob color
Add blob to global list
Are there any blobs in the global list that do not have been checked yet
yes
No
Are there any pixels in the blob that have not been checked?
yes
No
Calculate pixel value based on bitmasking
Create graphs based on pixel value
Connect created graphs to any existing graph in blob if possible
Are there any blobs in the global list that do not have been checked yet
yes
No
Is there more than 1 graph in the blob?
yes
No
Connect 2 graphs by adding nodes on an existing positions of graphs and merging
Launch OnFinish event
Blob
+ blobColor: int
+ blobName: string
+ pixelsInBlob:HashSet<vec2>
+ outlinePoints: List<vec2>
+ graphs: List<Graph>
+ positionsToCheck: HashSet<vec2>
+ positionsChecked: HashSet<vec2>
Graph
+ startNode: Node
+ endNode: Node
+ nodes: List<Node>
Node
+ nodePosition: Vec2

Scene settings

Camera height: This setting affects the height of the camera above the conveyor.

Camera focal length: This setting affects the focal length of the camera used to create images. Together with sensor size determines zoom.

Camera sensor size: This setting affects the sensor size of the camera used to create images. Together with focal length determines zoom.

Camera clip planes: This setting affects which objects will be rendered. The near clip plane prevents objects close to the camera from being rendered. The far clip plane prevents objects far from the camera from being rendered. The depth image is based on the percentage an object is between the near and far clip planes.

Pear spawn rate: This setting affects the frequency with which pears spawn on the conveyor.

Light intensity: This setting affects the brightness of the light.

Light direction: This setting affects the direction the light is coming from in euler angles.

Cast shadows: This setting affects if shadows should be cast.

Pear

Curvature X: This setting affects the amount of rotation the top of the pear can have around the X axis.

Curvature Z: This setting affects the amount of rotation the top of the pear can have around the Z axis.

Width: This setting affects the general width of the pear.

Width variation: This setting affects the width difference between pears.

Bottle shapedness: This setting affects how straight the pears are.

Pear top color: This setting affects the base color of the top of the pear.

Pear bottom color: This setting affects the base color of the bottom of the pear.

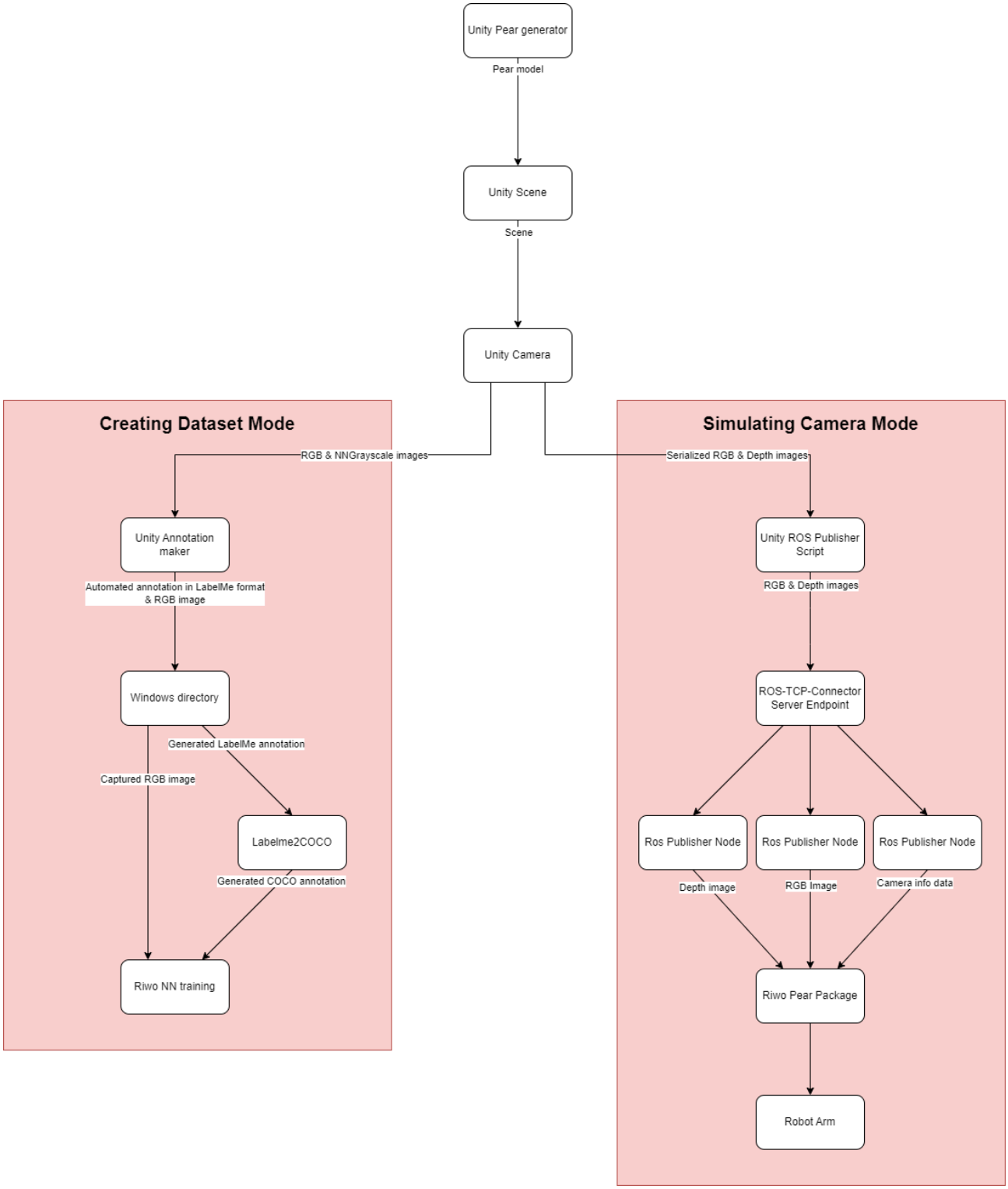
Brown area primary color: This setting affects the dominant color of brown areas and dots.

Brown area secondary color: This setting affects the secondary color of the brown areas and dots.

Brown area top density: This setting affects the amount of brown at the top of the pear.

Brown area bottom density: This setting affects the amount of brown at the bottom of the pear.

Scratch color: This setting affects the colour of scratches on the pear.


Unity Pear generator
Pear model
Unity Scene
Scene
Unity Camera
Creating Dataset Mode
RGB & NNGrayscale images
Unity Annotation maker
Automated annotation in LabelMe format & RGB image
Windows directory
Generated LabelMe annotation
Captured RGB image
Labelme2COCO
Generated COCO annotation
Riwo NN training
Simulating Camera Mode
Serialized RGB & Depth images
Unity ROS Publisher Script
RGB & Depth images
ROS-TCP-Connector Server Endpoint
Ros Publisher Node
Ros Publisher Node
Ros Publisher Node
Depth image
RGB Image
Camera info data
Riwo Pear Package
Robot Arm