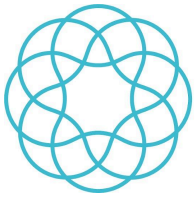
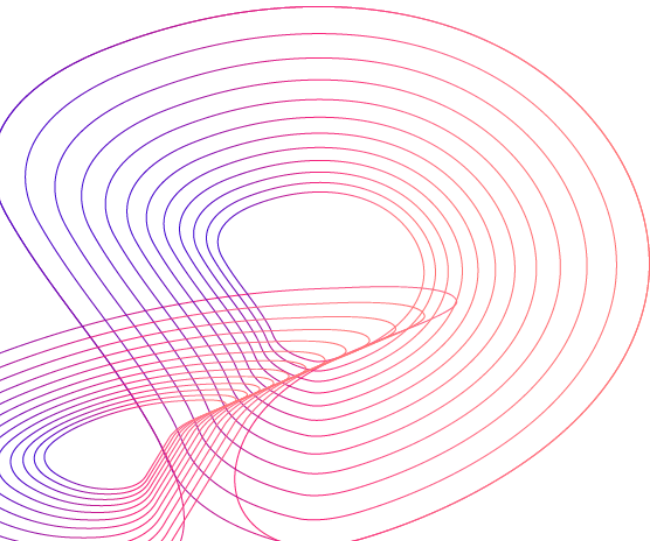


UNIVERSITY OF LIVERPOOL
MATHS SCHOOL
EDUCATION FOR 16-19 YEAR OLDS

3Dev

Jamie Andrews

Component: ¾ Programming Project
Course: H446 Computer Science
Candidate Number: 4003
Centre Number: 34409



Analysis

Problem Identification

Background

Computer generated images (CGI) are used across a number of industries such as film production, architecture and game development. In previous years, creating engaging and clear visuals was a task that had to be done manually - whether that be an architect sketching every last detail of a building or an animator drawing every single frame of an animation.

This greatly limited these industries for years, as vast amounts of time and money were spent on human labour in order to complete these tasks. This began to change with the introduction of computer graphics. From the late 1960s, methods such as ray-casting (1968) and z-buffering (1974) were developed to render 3D scenes, but these methods have their limitations in the complexity of the scenes they are able to generate. Early ray-casting only considered primary rays (rays from the camera) and not secondary rays such as reflections or shadows.

These features are essential for generating realistic scenes which closely resemble true textures and lighting, making it a viable technology for use in more professional industries.

More modern approaches such as ray tracing allow for these more complex effects and can also be optimised to render more detailed scenes without the need for such ridiculous processing times. However, many modern applications of ray tracing require expensive hardware which is not accessible to those who want to explore 3D visual industries in a recreational or semi professional capacity.

Therefore, I aim to create a system which can generate and render complex 3D scenes on modest modern hardware, using current ray tracing techniques. I will be focusing on optimizations which provide the best trade off between reduction in run time and simplicity of code development. This is because I plan on making the project open source in order to maximise accessibility for those with differing use requirements - the more straightforward the code, the easier it is for someone to specialise it for their particular use case.

Computational Methods

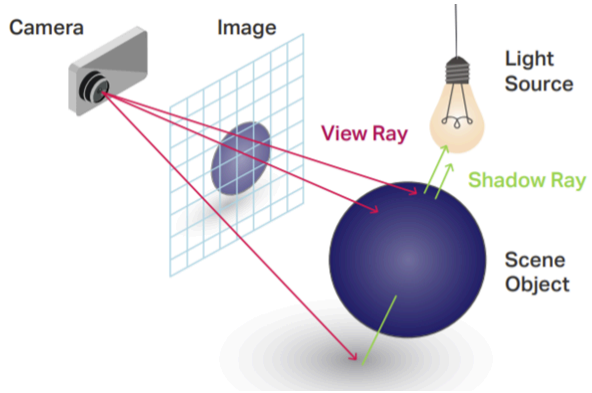
Ray tracing works by calculating the path of a light ray from the camera centre (eye point) to objects in a scene, through every pixel in the frame being rendered. This means performing lengthy vector calculations to find what objects each ray intercepts with and what colour value that ray represents for every single pixel. To do this without the use of a computer would take countless man hours in order to render just a simple, low resolution image.

Each ray can be thought of as a function, $P(t) = A + bt$, where P represents a 3D position along a line, A is the origin of the ray, and b is the direction of the ray. t represents a point in time, different values of t correspond to different positions along the path of the ray. Rays will be represented in code using a ray class, where A and b are vector objects, defined by its own class, and attributes of the ray class.

We now have to do 2 things to render an image:

- ➔ Calculate the ray through the pixel, from the eye point
- ➔ Compute the intersection points of the ray with objects in the scene

For this I will need to setup a camera (located at the origin) and viewport (through which the rays will pass into the scene)



One of the primary advantages to using a computer for this kind of task (aside from the fact that each individual calculation can be performed orders of magnitude times faster than the attempt of any human) is that, using modern hardware such as a GPU, methods such as parallel processing can be utilised to process many rays in the scene simultaneously. This is because the processing for the image can be broken down into a large number of rays, which can be calculated completely independently of each other - since they are not dependent on the behaviour of other rays. Additionally, since the same kind of calculations are being run for every ray, the **Single Instruction Multiple Data (SIMD)** architecture of modern GPUs lends itself to the simultaneous calculation of rays at a much faster rate than that of a CPU, which is forced to perform calculations for each pixel much less concurrently due to its limited number of cores.

When calculating the intersections between rays and objects, one option is to just have an array full of objects in the scene, all with their own different attributes. It is preferable to abstract all of the objects into a parent class which represents anything that can be hit by a ray. This means that later on in development it becomes easier to implement functionality around other types of hittable entities such as volumes. Additionally, we can have an abstract material class with attributes to describe how a ray is scattered once hit, which encapsulates the unique behaviour of different material types. This is opposed to having a universal material type with many different attributes that can be acknowledged or ignored for materials which have different properties.

In order to create smoother transitions between objects in the foreground and background without dramatically increasing the resolution, I can make use of antialiasing which takes multiple samples for each pixel and calculates an average for the colour. This is ideal for a computational approach as it involves the integration of a continuous function of light which falls onto a discrete portion of the image - something that cannot realistically be done by humans for detailed shapes. The most straightforward approach to this is to sample 4 points within the region bound halfway between the pixel and its 4 neighbours in each direction.

Overall, my project is amenable to computation methods because it is contingent on the processing of copious amounts of the same type of complex calculation. This is something the human brain is ill equipped for as we are not adept at thinking concurrently when performing long, complicated tasks such as vector calculations - unlike modern computing hardware that can handle much larger sets of tasks at a consistent rate.

Stakeholders

Stakeholder 1

My first primary stakeholder for this project is **Martin Linklater**, a game developer with over a decade of experience on mainstream 3D games such as Sea Of Thieves.

Martin needs something which can render 3D scenes, involving a range of features such as:

- ➔ Varieties of textures
- ➔ Volumes (such as smoke, fog or clouds)
- ➔ Varieties of shapes
- ➔ Lights
- ➔ Support for large quantities of objects

He plans to combine my scene renderer with animation software to create semi-realistic, cinematic cutscenes for his 3D indie game. This system would be suitable for his use case as he will need to make use of more realistic rendering effects on his personal computer.

My software should be a useful tool in the creation of his game and should not be the reason for any bottlenecks in the speed of his development. This means my ray tracer must be able to process and render a scene in reasonable working time so that Martin can swiftly move onto tasks which depend on it.

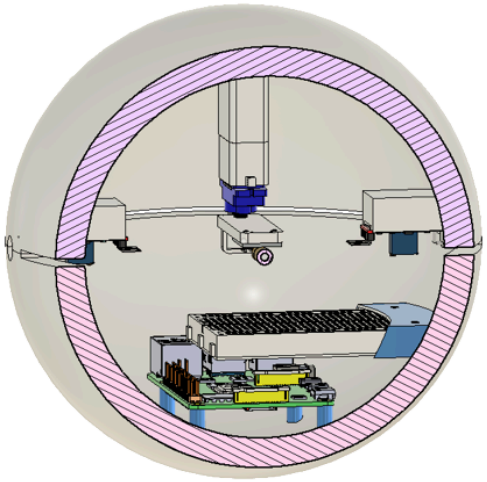
Martin will need to construct a variety of different scenes for different points throughout his game. This means one of the essential features for this project is an intuitive interface in order to create, save, and customise different scenes. Martin will then be able to easily keep track of his different cutscenes and edit them where necessary as the development of his game progresses.

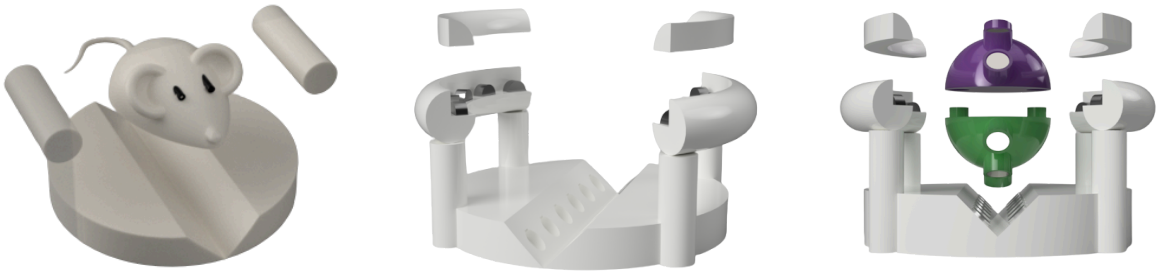
Martin is a developer who works on large professional games but also enjoys working on smaller scale personal games. My project will attempt to elevate the quality of his personal game graphics closer to that of his larger scale endeavours.

Throughout the development of my project Martin will be interviewed 2 times - once during the research stage of my project (To clearly outline what features he needs for the software to meet his needs), and once at the end of my project during the evaluation stage (After he has completed the final black box testing of my finished product). Martin will also be doing user acceptance testing after the rendering sprint of my project to provide feedback on the quality of the ray tracing and the sufficiency of the features offered by my renderer to meet his needs.

Stakeholder 2

My second stakeholder is **Jason Wang** who is an A-Level student with a passion for electronics projects (which he models and 3D prints plastic parts for) and making organic 3D models.





Jason needs a program which he can use to combine shapes to make basic prototypes for his electronic project models. He will then render these to get a preview of what the part will look like before modelling it fully in Fusion360. Additionally, Jason needs to go through a similar process for his organic model projects to get an idea for how complicated the shapes and parts of his model are likely to be, in order to decide whether or not the project is feasible.

Jason needs to be able to render a variety of materials which he might decide to use to build his components out of. He would also like to see the prototypes for his organic models rendered in as much detail as possible as he has invested in a high quality monitor for these projects. This means that 4k rendering support would be necessary to fully cater to his use case.

My project is very suitable for Jason's needs as I will be focusing on the ray traced renders as the primary feature of my project with a simple scene creator tool to make simple composite shapes and scenes.

Jason is also going to be interviewed during the research stage of my project, in order to identify the needs for his specific use case, and during the evaluation stage, to give a review on how well the software served his requirements. Additionally, Jason will be doing user acceptance testing like Martin but after the scene creation sprint so that he can give feedback on how successfully he was able to put together objects and scenes for his prototypes.

General Stakeholders

Due to the computationally intensive nature of ray tracing and its prominent use of parallel processing, it is necessary for the user to have a dedicated graphics card installed into their computer. However, due to the graphical nature of many industries which my software will apply to, this is a criterion that the majority of my prospective clients will already satisfy.

The target for my software are independent workers/freelancers as well as people who are passionate about exploring 3D graphics. Martin & Jason are going to represent people from my target audience in order to give insight into how my system is used by someone developing 3D graphics for an individual project.

However, I will also be providing my software to a variety of individuals - such as friends, family and other students - in order to perform alpha testing after the final user interface (& version control) sprint of my project. This allows me to gain a wider range of feedback on the work done in the other two sprints, as well as an idea of how intuitive my user interface is for beginner users who don't necessarily have much experience with other 3D graphics softwares.

Research

Existing Tools

Professional level software used for 3D image rendering includes tools such as **Blender[1]**, **D5-Render[2]** & **Fusion 360[3]** which all have slightly different advantages in different areas - making them more suited to different industries.

Scene Analysis

D5-Render is predominantly used for real world modelling such as architecture and open world landscapes or natural environments. I will review examples for both of these as well as some additional features which the platform offers.

D5-Render example scenes

There are 4 main features of this render that have a huge impact on the image:

- ➔ **Water Reflection**

The reflections of the forest in the stream are darker than the environment itself which helps make the water easily distinguishable from the objects around it. This makes the image look more realistic because in real life, some of the light is refracted through the water rather than all of it bouncing right off the surface. Additionally, there is a good balance between the clarity and blur of the reflection - since the surface of water in real life has ripples and imperfections that distort the reflection, while still being more clearly reflective than other non-matte surfaces such as metal.

- ➔ **Direct and indirect lighting**

This scene makes extensive use of shadows and rays of sunlight through the trees to highlight key areas of the image such as the animal drinking from the lake. However, there is also use of the indirect blue light coming from the rock, which does not heavily affect much of the scene around it other than the water but does contribute to the overall feeling of the environment.

D5 also offers other lighting effects, as described on their feature list[4] as follows:

AO Mode

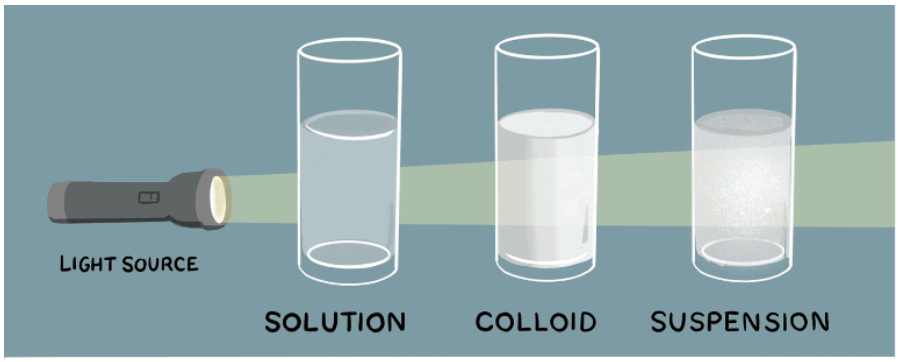
In AO mode, we can adjust the Ambient Occlusion intensity for both preview and output. It helps improve the realism of your rendering, bringing a sense of space and artistic contrast to the scene.

Ambient occlusion involves calculating how exposed each point on the surface of an object is to ambient light present throughout the scene in order to make shadows appear more smooth and realistic.

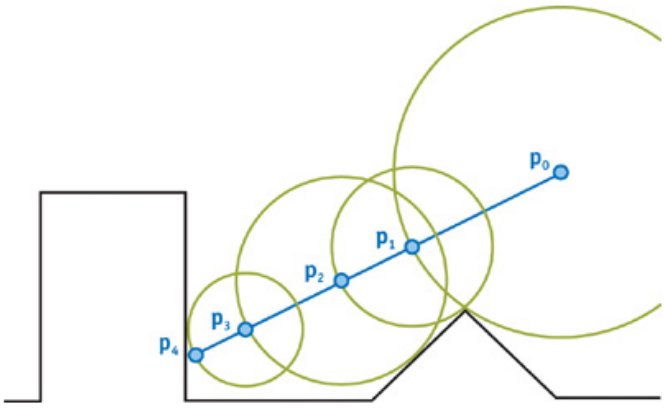
Volume Light

Utilizing Ray Marching in the ray tracing process, sampling the ShadowMap to simulate the realistic Tyndall effect.

The **Tyndall effect**[5] is the scattering of light from fine particles suspended in a fluid which makes the light more visible as it passes through.



D5 achieves this by making use of **ray marching**[6] which is a rendering technique that uses distance functions between a point and objects in a scene to determine how far a ray can travel without hitting an object. This is done by monitoring the minimum distance between the leading point on the ray with all the objects in the scene until it reaches a sufficiently small value.



From GPU Gems 2: Chapter 8.

➔ **Fog (Volumes)**

There is an increasing gradient in the thickness of fog noticed in the image with depth, this draws more attention to the foreground and the key features of the scene.

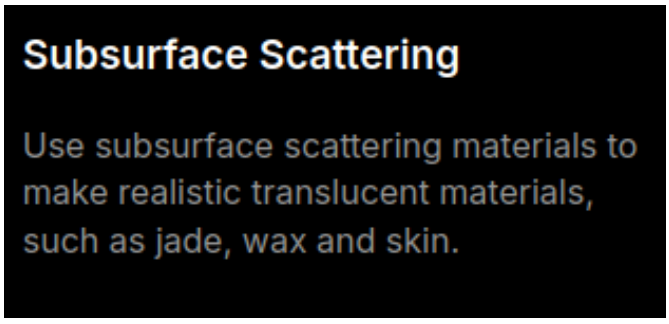
➔ **Wide range of models**

As well as all the effects and rendering techniques, a crucial factor which made this scene possible is the wide array of 3D models the designer was able to incorporate into the scene in order to give it character and make it more interesting to keep looking at for longer. It is important that users of my software are not bottlenecked in the uniqueness of their scenes by the limited number of shapes/models available. Since D5 is not primarily a modelling software (unlike the other two competitors on the list), it has large amounts of preset models available as well as the ability to import your own 3D model files into the program. I believe this importing of models will be most beneficial to my software as it prevents a bias towards one particular industry due to unbalanced support for different kinds of scenes which use different models.

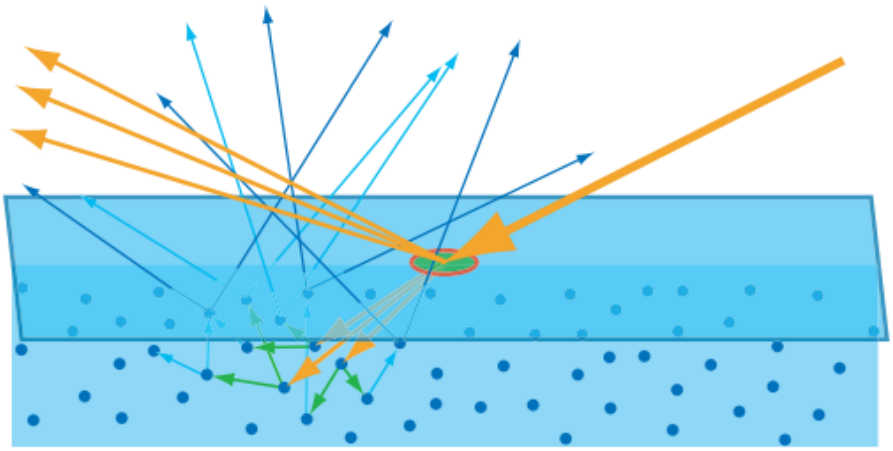
Similarly to the previous one, this image makes use of harsh lightning and shadows to create the desired aesthetic tone for the scene. This is something that I wish to recreate in my software by allowing the user to change the intensity and colour of light sources to fit the theme of the scene best. This scene also puts a large emphasis on textures, as both the buildings and general infrastructure visible are composed of a set of different materials. Because my user base is fairly open towards people from different industries, I think it would be equally useful to incorporate **importable textures** (as well as implementing a **texture mapping** system) in the same way I will include imports for 3D models.

Feature Review

D5 Also includes a range of other features which I plan to fully or partially recreate:



Subsurface scattering[7] would be useful in my software to render more realistic looking skin as some users (such as my primary stakeholder Martin) will want to render characters in high definition cinematic scenes. This technique works to render translucent materials by reflecting some rays off the surface of the material and diffusely scattering the rest after they pass through a portion of the object.



Cinematic Camera Effect

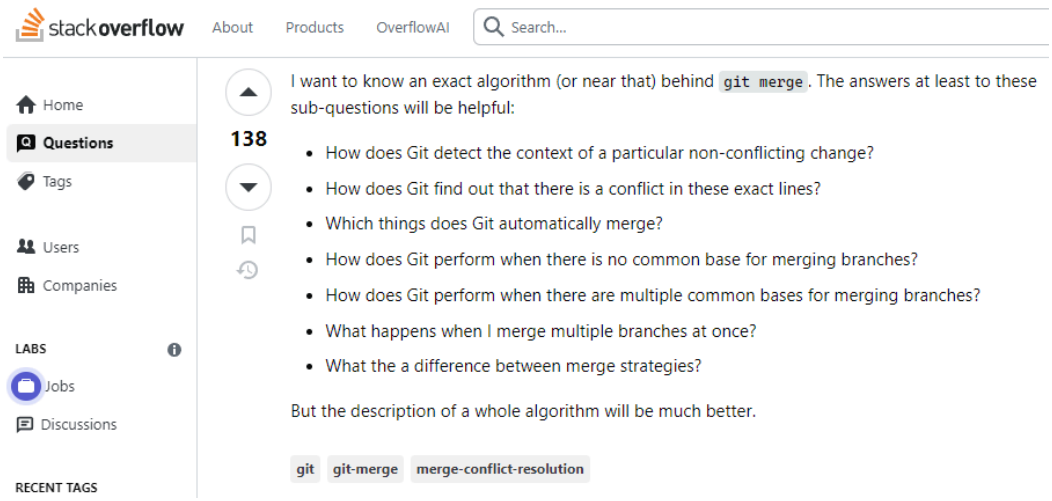
Create a cinematic scene through parameters including Bloom, Lens Flare, Chromatic Aberration, and Vignette.

Cinematic camera effects such as lens flare may be beneficial for some cases but are often utilised more in video rendering as it can draw too much attention away from the focus of the image unless only visible for a short period of time.

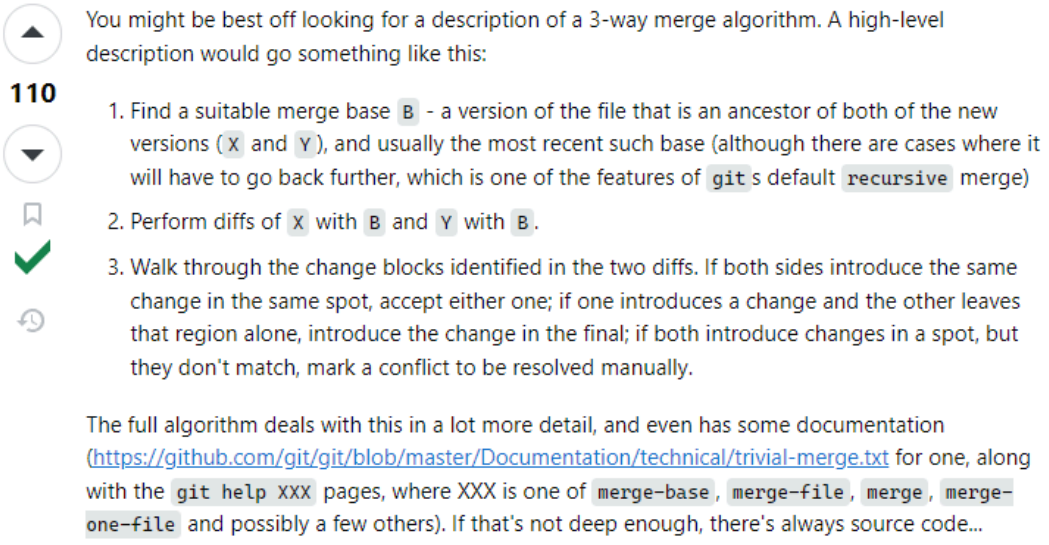
Merge Projects*

Teamwork made easy. Support merging up to 10 projects at the same time.

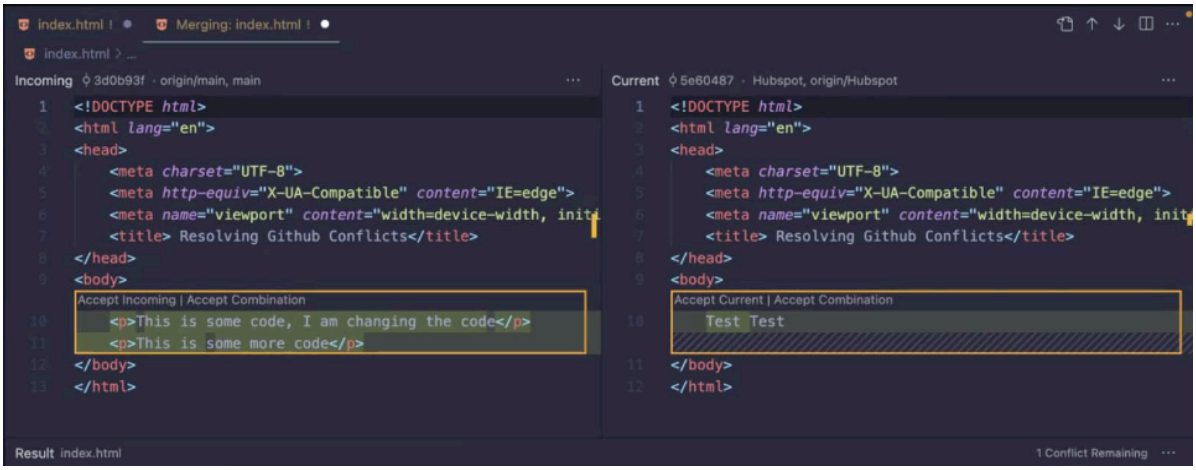
There may be teams of developers who are working on different aspects of a complicated scene and want to work independently of separate sections before collecting their work into one scene. **Merging multiple projects** across different accounts (or the same account) allows users to do this without having to use other third party software.



This merging feature is reminiscent of the merge feature for github repositories so I explored a stack overflow forum post to identify the main steps which a merging algorithm would need to carry out in order to produce a functional file which combines the two projects.



For any objects which overlap when merged, I plan to recreate the github merge conflict flag by highlighting conflicting objects in a visible bright red colour and introducing a conflict tab with a list of all the conflicting object sets for the user to go through and select the object(s) they wish to keep and objects they wish to discard.



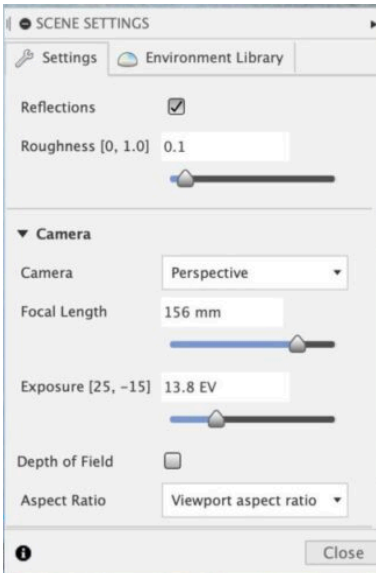
Stakeholder Interviews

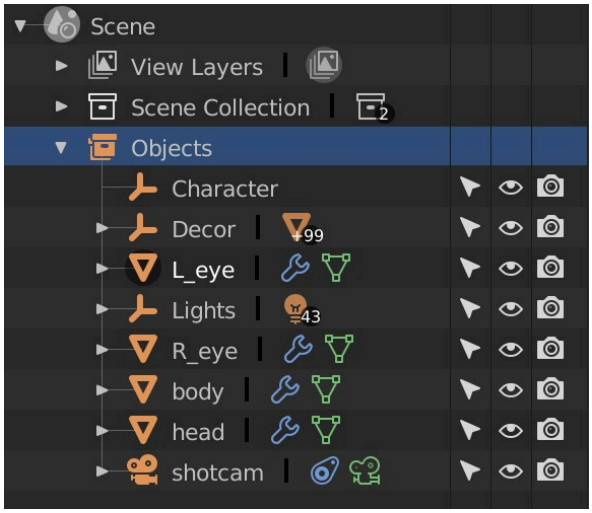
General Experience

➔ **What specific features do you find most useful in Blender and Fusion 360?**

- *"Render sliders (nearly all of them have it) just allow you to select for what refinement of renders you want, and having a slide made it very easy."* - Jason

- *"I find the object management tab in blender very useful to organise everything I have in my scene - particularly small objects which can be easy to lose track of amongst the rest of the objects"* - Martin





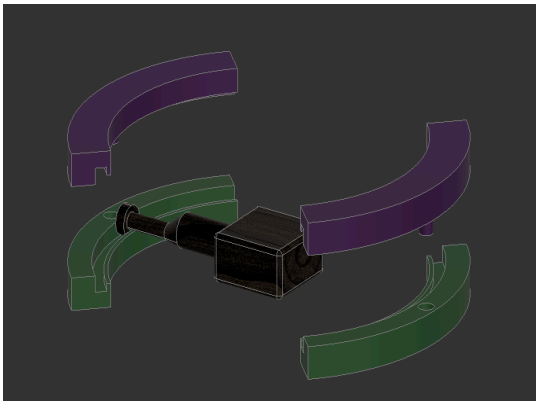
➔ **Are there any features or tools you wish these software packages had?**

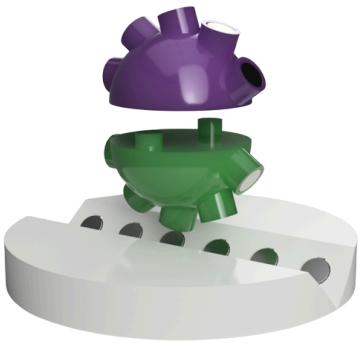
- *“Blender covered everything i needed for organic models and Fusion 360 covered everything for solid or mechanical things”* - Jason
- *“I think these big software packages pretty much cover everything anyone could want to make, the only potential additions would be for really niche use cases that I don't need.”* - Martin

Rendering Needs

➔ **What types of scenes or objects do you typically render?**

- *“Still plastic parts I 3D print for electronics projects or organic models like animals just for fun”* - Jason





- *“Mostly putting together open scenes with a few characters and props”* - Martin

➔ **What are your primary goals for rendering quality and speed?**

- *“I don’t have many time restraints, I am more interested in the render being as high quality as it can be”* - Jason
- *“Time isn't an issue for still images its just quality I care about”* - Martin

  I will prioritise generating the highest quality renders possible over the time it takes to render the final image (within reason).

Workflow Integration

➔ **What types of import/export capabilities are essential for your work?**

- *“I need to be able to import custom textures and I only really need to be able to save the final render as an image. It would be useful if I could choose the aspect ratio of the picture though.”* - Jason

  I will create a dropdown when rendering an image to decide the aspect ratio of the image - changing the aspect ratio will have no impact on the resolution of the render so that all options will produce the same image quality.

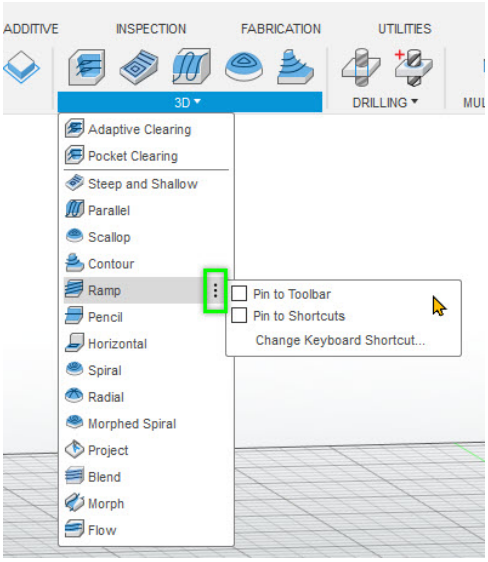
- *“Importing all the basic stuff like textures and backgrounds is all I need, as well as saving with all the usual image file types like jpg and png”* - Martin

  I will implement an import option for backgrounds and textures which will add a file from the users computer to the appropriate database. These imported textures/backgrounds can then be used for mapping in project files (as long as their files are the correct format).

User Interface and Usability

➔ **How important is the user interface to your workflow, and what aspects do you find most crucial?**

- *“These softwares often have a lot of features that can be hard to keep track of so I really like when it is very clear how to navigate to all of the key features”* - Jason
- *“In some blender dropdowns with lots of options, everything is crammed together with small writing which can be hard to navigate”* - Martin



I will try to follow Fusion 360’s dropdown menu style with spaced options and shorter naming styles. Fusion 360 also implements extensive use of small diagrams in order to illustrate the function of different buttons and options, this is something I may be able to take advantage of depending on the complexity of implementation.

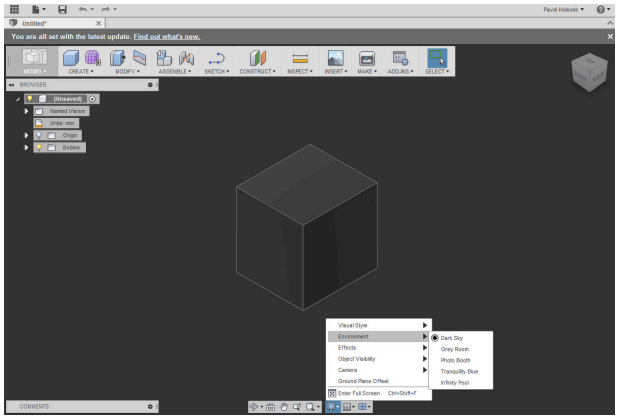
➔ **Are there any UI features in Blender or Fusion 360 that you find particularly effective or frustrating?**

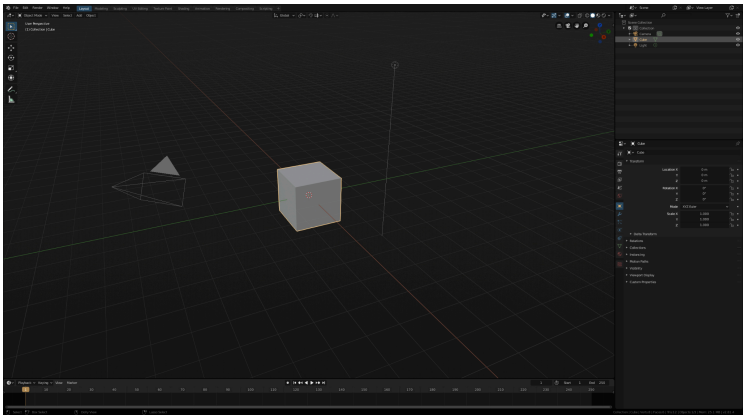
- *“There is a large database of textures which I find useful but how it's arranged isn't the easiest as it feels like everything is everywhere. The environment was also pretty cool but had the same issue with user experience”* - Jason

  Blender displays an extremely extensive database of textures in large lists that are often difficult to navigate through in order to find the texture you are looking for - especially if you don't know what it is named as. To combat this I will attempt to implement a searching feature to the textures database which allocates multiple descriptors to each texture which will be used for search referencing so that even if the user does not know the name of a texture they will still be able to search for it more easily.

- *“I work pretty late at night so having a dark mode is something I find pretty nice to have, the Fusion 360 dark mode isn't ideal though because only the environment is dark and not the tabs”* - Martin

  I will try to adopt a more blender-style dark mode rather than the default fusion 360 dark mode which does not affect the entire interface





Collaboration and Sharing

➔ **Do you often collaborate with others on projects? If so, what collaboration tools or features are important?**

- *“I don't really most of my projects are individual”* - Jason
- *“Yeah I work with people on my team pretty often on developing characters”* - Martin

I will allow multiple users to sign in on the same install of the program in order to work on separate projects. These users can then merge their project files, as mentioned under the “Existing Tools” section of my research.

➔ **How do you handle version control or scene management for complex projects?**

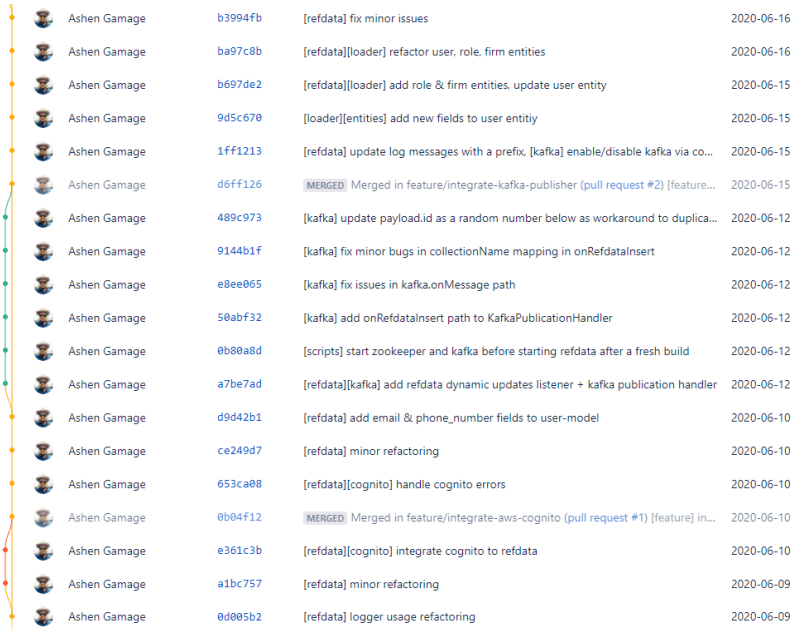
- *“I like being able to choose between saving projects as versions or overwriting the file, that way I don't lose old version of bigger more complicated projects but I also don't end up with lots of unnecessary files for a simple scene”* - Jason

I will give the user an option when saving project files if they wish to overwrite the current file or create a new file with the updated content in the same folder as the current file

- *“We use GitHub to manage the versions of our model files and which versions of the game code they were developed around”* - Martin

It may be beneficial to allow the user to save comments attached to each version of a project as they save it - similar to the Git commit message system. Following inspiration from Git, saving the date on which the version was saved could also be useful for 2 reasons:

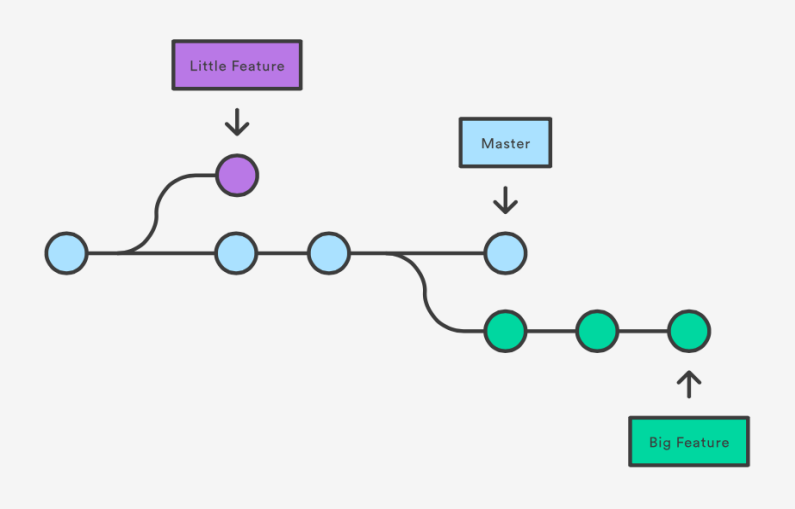
➔ Allowing the user to see the dates on which they saved a version lets them get a better idea of how on track they are with the progress they want to make on a project within certain periods of time.
➔ Automatically displaying the version options of a project in reverse chronological order when the user wants to open one.



For this, I will most likely need to keep a separate file to the main scene data file in the folder of each project in order to store meta-data such as date, time, owner user and commit message.

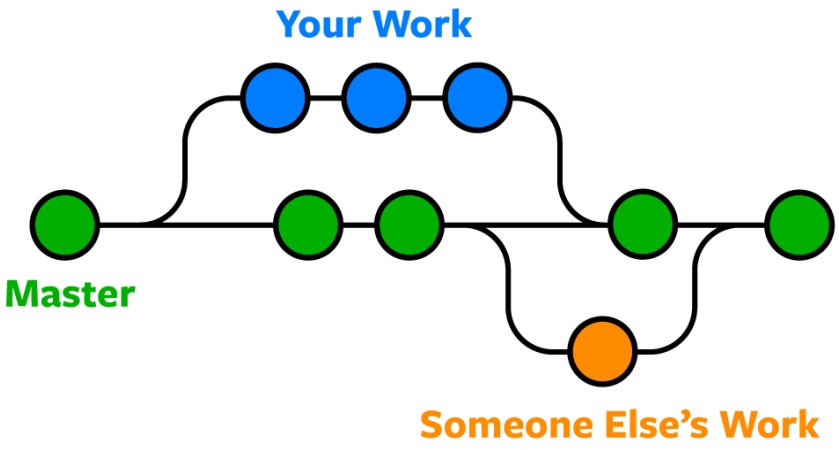
If a user decides they would not like to create separate version files at each save and would like to overwrite the data on the current file instead, they will still be able to save time/date stamped commit messages at each save but will not be able to go back and view what the project was like at that save.

Additionally, I would like to emulate the GitHub branching feature such that if a user goes back to a previous version save of a project and then edits it, they will be given the option to create a new branch originating at the current version or “chop off” the current branch which extends from this version (and replace it with the new changes) upon saving.



This feature would compliment the merging feature mentioned earlier in 2 use cases:

- ➔ Users can create branches for experimental additions and then merge back with the main branch once they are confident that they would like to add the changes to the project.
- ➔ When multiple users are working on the same project they can create different branches and then merge their versions together at a later stage in the project. This means that teams working collaboratively can keep all their work in one project folder instead of having many separate project instances and then merging all of them together.



Computation

I have read and understood the maths and theory from the Ray Tracing in One Weekend[8] book by Peter Shirley. This book outlines all of the computation required to create a ray tracer and the methods used to put them into practice. However, the code included in the book is C++ which I do not plan to use, so I won't be using any code from the book. Additionally, I plan to structure my program differently since the ray tracing portion of my software is going to be run by the user interface - so I will be taking a much more modular approach to my design. Finally, there are sections covered in the book that are largely specific to C++ so I have ignored those sections and taken mostly from sections explaining the relevant mathematics - which I will implement as best I can using my chosen language.

Throughout the development of my project, I plan to replicate the tests performed by the author of the book as he explains the stages of developing his ray tracer. This will help me ensure that I am staying on track and producing outputs to a suitable standard. However, this will primarily apply to functional white box tests, once I have all my functionality working I will perform user acceptance testing with more complicated scenes that myself and my stakeholders will create.

The primary developmental help I am taking from the book is the ppm (portable pixmap) image format. This is a text based image format which uses integers to define the rgb colour values (between a given range) of pixels on a grid (of given size). The image below from its wikipedia page[9] outlines this format.

PPM example [edit]

This is an example of a color RGB image stored in PPM format. (Not shown are the newline character(s) at the end of each line.)

```
P3
# "P3" means this is a RGB color image in ASCII
# "3 2" is the width and height of the image in pixels
# "255" is the maximum value for each color
# This, up through the "255" line below are the header.
# Everything after that is the image data: RGB triplets.
# In order: red, green, blue, yellow, white, and black.
3 2
255
255   0   0
  0 255   0
  0   0 255
255 255   0
255 255 255
  0   0   0
```

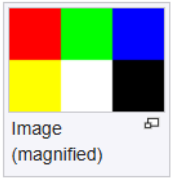
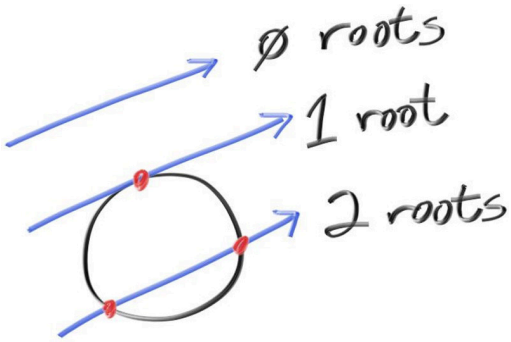


Image
(magnified)

Mathematics

During the majority of my development I will be working with sphere objects, and I will only be implementing quadrilateral intersection at a later stage if I have the time for it. Sphere-ray intersection is calculated by finding the roots of a quadratic equation generated using the vector of the ray and the sphere equation.



The following is an extract from the book explaining this intersection equation:

First, recall that a vector dotted with itself is equal to the squared length of that vector.

Second, notice how the equation for `b` has a factor of negative two in it. Consider what happens to the quadratic equation if $b = -2h$:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$= \frac{-(-2h) \pm \sqrt{(-2h)^2 - 4ac}}{2a}$$

$$= \frac{2h \pm 2\sqrt{h^2 - ac}}{2a}$$

$$= \frac{h \pm \sqrt{h^2 - ac}}{a}$$

This simplifies nicely, so we'll use it. So solving for h:

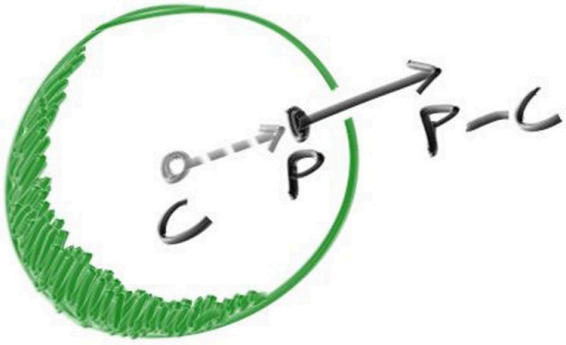
$$b = -2\mathbf{d} \cdot (\mathbf{C} - \mathbf{Q})$$

$$b = -2h$$

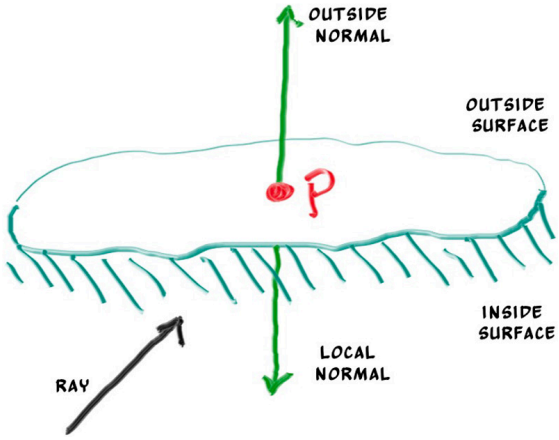
$$h = \frac{b}{-2} = \mathbf{d} \cdot (\mathbf{C} - \mathbf{Q})$$

Using these observations, we can now simplify the sphere-intersection code to this:

To implement shading, I will begin by calculating the surface normals of a smooth sphere and then change the implementation to act as if the surface is rough in order to scatter rays randomly, creating a more realistic effect.



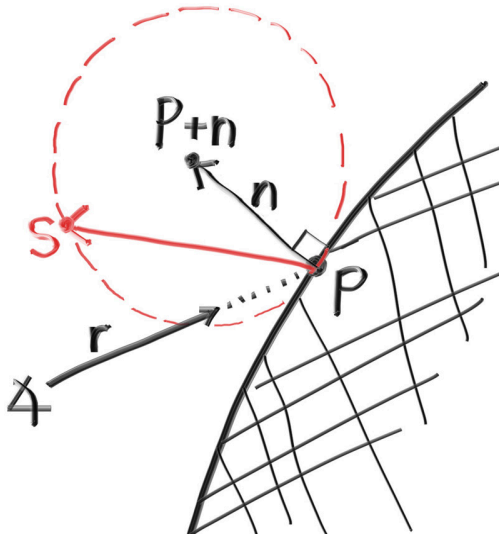
We need to perform operations when calculating surface normals to ensure that we don't get mixed up between the normal of the surface (at the point of intersection) on the outside of the surface and on the inside of the surface.



Diffuse materials will be rendered using a rejection method to manipulate random vectors so we only get results on the surface of the hemisphere. Rejection methods produce random vector samples repeatedly until we find one which meets desired criteria - there are many complicated ways of setting out rejection method ciretia but I will be using the following three:

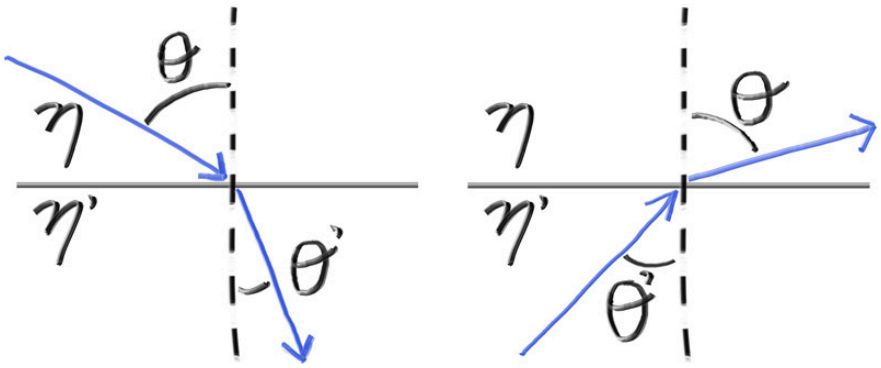
1. Generate a random vector inside the unit sphere
2. Normalize this vector to extend it to the sphere surface
3. Invert the normalized vector if it falls onto the wrong hemisphere

We can then check the sample is on the correct side of the hemisphere by comparing it with the surface normal.



The proportion of a ray's colour it retains after reflecting off an object is defined by the object's colour, i.e. if a ray retains 100% of its original colour then the object is white and if it retains none then the object is completely black. The proportion of incoming rays which are kept after bouncing into a surface will define how bright the surface is.

Specular reflection (for smooth/reflective objects) is calculated using snell's law



In order to determine the direction of the refracted ray, we have to solve for $\sin\theta'$:

$$\sin\theta' = \frac{\eta}{\eta'} \cdot \sin\theta$$

On the refracted side of the surface there is a refracted ray $\mathbf{R}'$ and a normal $\mathbf{n}'$, and there exists an angle, θ', between them. We can split $\mathbf{R}'$ into the parts of the ray that are perpendicular to $\mathbf{n}'$ and parallel to $\mathbf{n}'$:

$$\mathbf{R}' = \mathbf{R}'_{\perp} + \mathbf{R}'_{\parallel}$$

If we solve for $\mathbf{R}'_{\perp}$ and $\mathbf{R}'_{\parallel}$ we get:

$$\mathbf{R}'_{\perp} = \frac{\eta}{\eta'}(\mathbf{R} + \cos\theta\mathbf{n})$$

$$\mathbf{R}'_{\parallel} = -\sqrt{1 - |\mathbf{R}'_{\perp}|^2}\mathbf{n}$$

You can go ahead and prove this for yourself if you want, but we will treat it as fact and move on. The rest of the book will not require you to understand the proof.

We know the value of every term on the right-hand side except for $\cos\theta$. It is well known that the dot product of two vectors can be explained in terms of the cosine of the angle between them:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$

If we restrict $\mathbf{a}$ and $\mathbf{b}$ to be unit vectors:

$$\mathbf{a} \cdot \mathbf{b} = \cos\theta$$

We can now rewrite $\mathbf{R}'_{\perp}$ in terms of known quantities:

$$\mathbf{R}'_{\perp} = \frac{\eta}{\eta'}(\mathbf{R} + (-\mathbf{R} \cdot \mathbf{n})\mathbf{n})$$

Total internal reflection is used to simulate accurate reflection since different portions of rays are reflected and refracted as they hit a surface depending on the material and the angle of incidence.

$$\sin\theta' = \frac{\eta}{\eta'} \cdot \sin\theta$$

Vectors will be used to define the orientation of the camera based on its position and the position of the object it is focusing on. These values can be altered to change the viewing direction of the camera and the subsequent image produced when rendered.

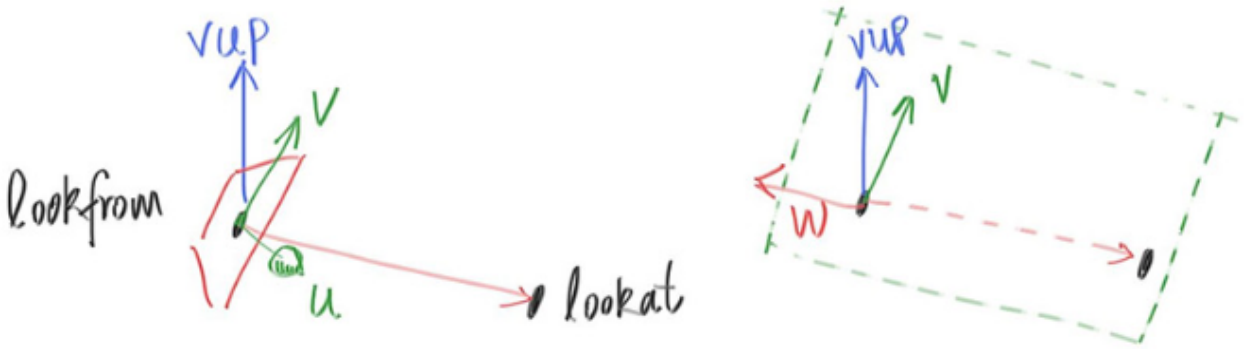


Figure 20: *Camera view up direction*

In order to make the image more realistic (as if taken by a physical camera) , defocus blur will be implemented by setting a plane on which the camera will focus - in order to simulate depth of field and focal length. The space in front of and behind this plane will be slightly blurred out of focus.

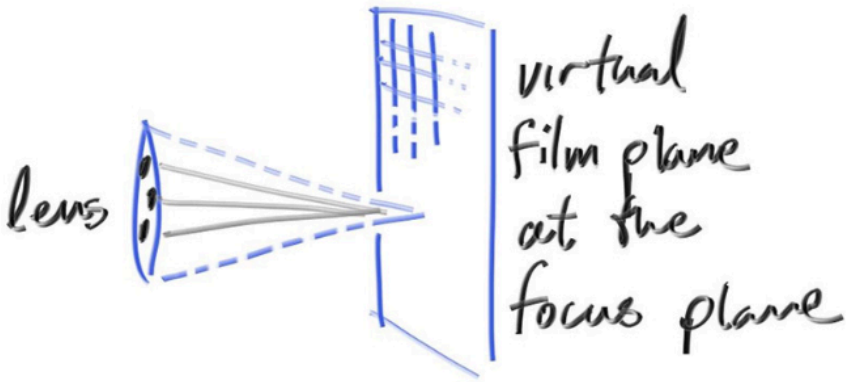


Figure 22: *Camera focus plane*

Finally, anti-aliasing will be implemented to simulate the way our eyes perceive smoother transitions between objects and their background. This is done by taking multiple pixel samples around a point and averaging the colour values of the objects they hit in order to make edges appear less harsh.

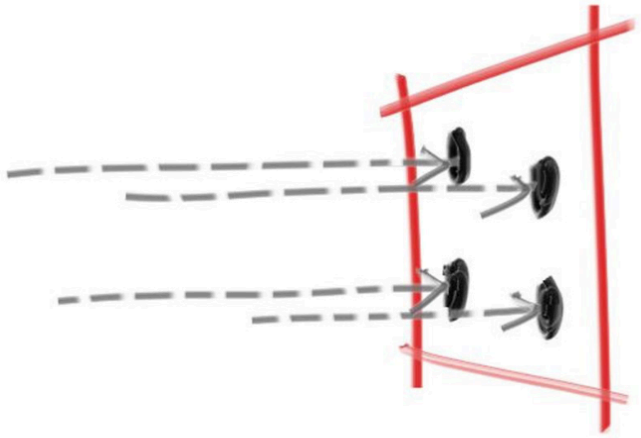
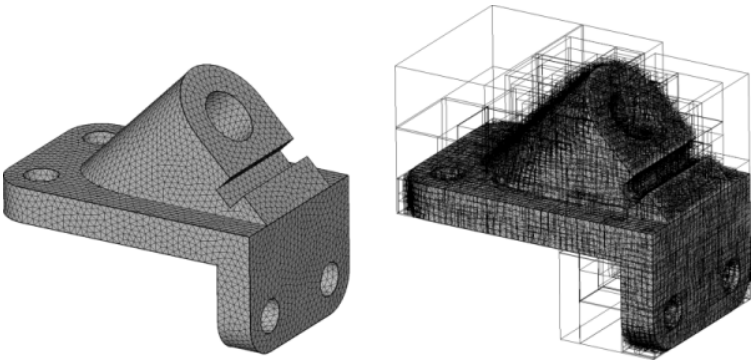


Figure 8: *Pixel samples*

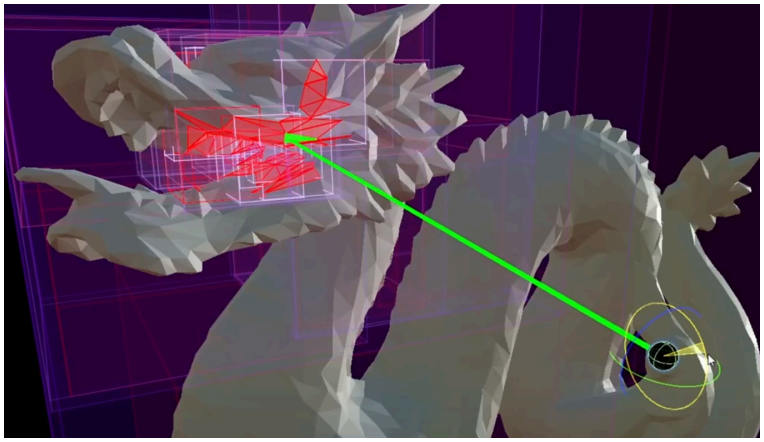
If I have time after covering these features, then I plan to read the next edition of the Ray Tracing in One Weekend book, which is called Ray Tracing: The Next Week[10]. This will cover more features which aid in making images appear more realistic and adding special effects.

In order to optimise the process of deciding whether or not a ray intersects an object, the book references bounding volume hierarchies (BVH). During my research I have also used a video by Sebastian Lague[11] to aid my understanding of this concept.



BVH split models into sections bounded by cuboids which themselves are then split into more smaller cuboids. Rather than checking every triangle of a mesh for ray intersections, BVH checks which of these larger cuboids the ray intersects first and then discards all of the calculations for the cuboids we now know it does not intersect with. This helps reduce the total number of intersection calculations by narrowing down the search area as the ray hits the bounding boxes

The visual below from Sebastians video shows a visual example of the search area of mesh triangles being reduced to just a few by the ray passing through a group of cuboids first. It then checks this smaller set of mesh triangles to find exactly which one intersects with the ray.



Colour Theory

Study 1: Mehta and Zhu (2009)[12]

This foundational study investigated how red and blue influence cognitive performance on tasks requiring either attention to detail or creativity, across six experiments.

Methodology

- **Participants:** Over 600 participants across six experiments.
- **Experimental Setup:** Tasks were conducted on computers with red, blue, or white screen backgrounds. The use of computers to expose participants to the light is suitable for generating valid conclusions for my project, since it is a computer based application.
- **Tasks:**
    - Creative tasks - brainstorming, analogical reasoning, and toy design.
    - Detail-oriented tasks - proofreading and recall.

Findings

1. Creativity

Participants that were exposed to blue backgrounds generated significantly more creative ideas than those exposed to red or white backgrounds. For example, in a task where participants asked to design toys using geometric shapes, those working with blue shapes produced more original and novel designs - compared to less innovative practical designs created by those under red background conditions.

- Blue invoked an "approach motivation," which encouraged exploratory and open-ended thinking.

2. Cognitive Mechanisms:

- Blue is often associated with peace, openness, and calmness (e.g. the sky or the ocean). This fosters a safe environment conducive to risk-taking and innovation.
- In contrast, red triggered "avoidance motivation," which is linked to vigilance and caution due to its association with danger (e.g. stop signs on roads).

Study 2: Zhu et al. (2009)[13]

A study at the University of British Columbia built on Mehta and Zhu's work by further exploring the effects of blue and red on cognition - in order to reconcile conflicting findings in their prior research regarding the cognitive effects of color - by examining how task type influences performance under different color conditions.

Methodology

- **Participants:** Over 600 participants from 2007 - 2008.
- **Experimental Setup:** Participants completed six cognitive tasks under red, blue, or white computer screen backgrounds.
- **Tasks:**
    - Creative tasks - brainstorming and solving analogies.
    - Detail-oriented tasks - memory recall and proofreading.

Findings

1. Creative Outputs:
Blue environments led to double the creative output when compared to red environments. For example, participants brainstorming under blue conditions generated more diverse ideas than those given a red background.

2. Psychological Associations:
Blue influenced feelings of tranquility and/or openness due to its associations with nature, such as water and the sky. These associations encouraged "thinking outside the box," which is essential for creativity.

3. Contrast with Red:
Red was more effective for detail-oriented tasks and impaired performance on creative tasks due to its association with danger, errors, and caution.

4. Unconscious Effects:
Participants generally were not consciously aware of how colors influenced their performance. However, their cognitive responses aligned with the motivational states mentioned previously.

Implications for Creativity

Both of these studies back up the idea that blue encourages creative thinking via its psychological association with safety:

- Blue creates a calming environment - reducing stress and mental constraints.
- This promotes divergent thinking - essential for generating new ideas - by encouraging exploratory thinking.
- The "approach motivation" triggered by blue enables individuals to take more mental risks without any subconscious fear or worry.

These findings are particularly relevant for environments where innovation plays a large role, such as 3D design software.

Application in 3D Design Software

A 3D design app is a tool designed for creativity. Users utilise this software for conceptualising models, experimenting with designs and arranging scenes. The insight from these studies provide a strong justification for applying a blue interface to this kind of application:

How Blue Benefits 3D Design Users

1. Fostering Divergent Thinking:
A blue interface can encourage the user to explore unconventional designs in e.g. concept art or architectural visualisation.

2. Enhancing Productivity:
As shown in Zhu et al.'s study, participants exposed to blue produced twice the creative outputs as those exposed to red. This directly translates to greater productivity in iterative design processes such as model prototyping.

3. Reducing Cognitive Fatigue:
Long design sessions can be mentally exhausting as precise designs require lots of concentration. Blue's calming effects can reduce stress during extended use of the software, maintaining focus and creativity over time - also improving user retention for the app.

4. Encouraging Risk-Taking:
The "approach motivation" which is induced by blue can support risk-taking behavior. This is hugely important for innovation to take place in industries like gaming, film production, or product design.

5. Improving User Experience:
Aesthetically appealing applications are a significant factor in user satisfaction. Well-designed blue interfaces not only enhance functionality but also creates a more enjoyable and easy time for the user.

Interface Design Recommendations

To maximize the benefits of blue in a 3D design application:

- I plan to use shades of blue for my primary interface elements in menus, toolbars, and background themes.
- Incorporating complementary colors like white or gray for contrast, while maintaining visual harmony, may be useful for aesthetic improvements.
- I will avoid overly saturated blues which may cause strain on the eyes; softer tones inspired by natural elements will be much more preferable.
- The use of gradients or subtle animations in blue can evoke tranquility without distracting users from their work, and therefore maintaining their productivity.

Broader Implications

These findings extend to software design in other creative industries:

- Education: Blue-themed learning tools can enhance students' creativity, this links well with my application as one of its intended uses is for people learning 3d design development.

- Workplaces: Offices painted in shades of blue can inspire innovative thinking. This also links well for my app as it should meet industry quality standards and be used within the workplace.

Conclusion

The studies by Mehta & Zhu (2009) and Zhu et al. (2009) provide compelling evidence that the color blue enhances creativity by fostering an environment of openness, tranquility, and exploration. These insights are directly applicable to the development of 3D design applications where creativity is paramount. By adopting a blue-themed interface grounded in scientific research, developers can create tools that not only support but actively enhance users' ability to innovate—making them more productive while ensuring an enjoyable user experience.

References

1. Blender Foundation (2019). *blender.org - Home of the Blender project - Free and Open 3D Creation Software*. [online] blender.org. Available at: https://www.blender.org/.
2. www.d5render.com. (n.d.). *D5 Render | Real-Time Ray Tracing 3D Rendering Software*. [online] Available at: https://www.d5render.com/.
3. Autodesk (2021). *Fusion 360 | 3D CAD, CAM, CAE & PCB Cloud-Based Software | Autodesk*. [online] Autodesk.com. Available at: https://www.autodesk.com/products/fusion-360/overview?term=1-YEAR&tab=subscription.
4. D5render.com. (2020). *Feature | D5 Render*. [online] Available at: https://www.d5render.com/features [Accessed 9 Sep. 2024].
5. Wikipedia. (2021). *Tyndall effect*. [online] Available at: https://en.wikipedia.org/wiki/Tyndall_effect.
6. Zero Wind :: Jamie Wong. (n.d.). *Ray Marching and Signed Distance Functions*. [online] Available at: https://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/#the-raymarching-algorithm.
7. therealmjp.github.io. (2019). *An Introduction To Real-Time Subsurface Scattering*. [online] Available at: https://therealmjp.github.io/posts/sss-intro/.
8. Github.io. (2019). Available at: https://raytracing.github.io/books/RayTracingInOneWeekend.html.
9. Wikipedia. (2022). *Netpbm*. [online] Available at: https://en.wikipedia.org/wiki/Netpbm.
10. Github.io. (2023). Available at: https://raytracing.github.io/books/RayTracingTheNextWeek.html.
11. Sebastian Lague (2024). *Coding Adventure: Optimizing a Ray Tracer (by building a BVH)*. [online] YouTube. Available at: https://www.youtube.com/watch?v=C1H4zIiCOaI.
12. Mehta, R. and Zhu, R. (2009). Blue or Red? Exploring the Effect of Color on Cognitive Task Performances. *Science*, [online] 323(5918), pp.1226–1229. doi:https://doi.org/10.1126/science.1169144.
13. University of British Columbia (2009). *Effect of colors: Blue boosts creativity, while red enhances attention to detail*. [online] ScienceDaily. Available at: https://www.sciencedaily.com/releases/2009/02/090205142143.htm.

Features & Limitations

Features

Settings Menu

Render Preferences

The “render preferences” section of the settings menu will have sliders to control preferences for the render. This will include the depth of field as well as the intensity of various attributes such as motion blur, lights and camera effects. This will allow the user to intuitively fine tune their render to achieve the look they are going for without having to know all of the exact values for the render preferences or re-type guesses until they find something which works. In practice, this helps people get renders finished quicker so they can move onto other areas of their project.

Additionally, users will have the option to toggle certain effects on and off which cannot be easily scaled. Doing this will allow users with different hardware capabilities to get scenes rendered more easily by removing some of the less impactful effects in their use case.

Finally, there will be two dropdowns available for the user to specify the desired resolution and aspect ratio for their render - changing the resolution will not affect the aspect ratio.

I do not expect this to take long to implement as once I have the basic slider/dropdown functionality which is easy to achieve with most standard ui libraries, all I need to do is create multiple instances of each of these for each preference and display them on a window. However, this feature is dependent on the implementation of the rendering functionality itself, as well as all of the effects to go with it which means I may only be able to implement it in a later sprint.

Gallery

Next, there will be a gallery section to the settings menu which will contain two subsections: a textures gallery and a backgrounds gallery. Both of these will display all the textures/backgrounds the user has downloaded, which they can import into any of their project folders, in a grid format - showing a small visual sample of each texture/background above its title.

The gallery will have a smart searching system which allows users to search for both textures and backgrounds without knowing their exact title. This will be accomplished by allowing the user to specify multiple categories which a texture/background fall into upon importation into their gallery database, these categories can then be used as search terms when looking for a specific import in order to narrow down the search. This becomes increasingly valuable as the number of files the user downloads into their gallery increases. The user will be able to access a number of materials such as glass, steel and marble by default as they can be generated without the use of texture mapping. There will be an option to export the gallery databases (either the textures, backgrounds or both) into a shareable format. When one user downloads and loads the project of another user, they will have to download the textures/backgrounds used in that

scene as part of the project file, they will then be given the option to either save these assets into their own gallery or keep them just inside the project file.

This will most likely be the most lengthy aspect of the settings menu to implement but I still do not think that it will take an extremely long time as all of the file handling code will be done using python which is my most confident language.

Scene Creator

The scene creator tool is the main interface that the user will be interacting with while using the software (There will be an option to view this interface in either dark mode or light mode).

Object Manager

This will be a collapsible tab at the side of the screen which will contain a list of all the objects added to the scene, along with all of their attributes on a collapsible dropdown (each attribute will have a slider for the user to easily edit the object). Each object will have a visual aid next to its title, showing a small copy of what the object looks like.

Object Addition Menu

When the user wants to add an object to the scene, a window will appear in the middle of the screen allowing them to choose the type of object to add and change its default attributes if they want to.

Importing Assets

Users can import backgrounds/textures from their gallery into a project scene and then apply them to the scene/objects. I hope to be able to implement a drag and drop feature for this but I may need to just create a small pop-up menu window when applying assets. The drag and drop mechanic is the most common way of implementing this kind of feature so it would be easiest for less experienced users to understand. However, this is more complicated so if I am running low on time I may not be able to complete it.

I am going to have to validate imported files before trying to apply them to scenes or objects. This can either be done at the point of application or when downloading into the gallery. I think it is most advantageous to validate at the point of download so that if the user wishes to use the same assets in multiple projects they can do so without having to perform redundant validations.

Scene Previews

I would like to implement a 3D visual preview of the scene that is going to be rendered as the user is adding and editing objects in the scene. This would make it far easier for the user to see what their scene is going to look like and make adjustments without having to continually render and re-render their scene. This may be fairly difficult to implement depending on the capability of the libraries available to me but given the profound impact the feature would have on project development I think it is worth spending the time to find a way to successfully implement it.

Rendering Methods

If a user has lots of scenes they would like to render in high detail and with lots of computationally intensive effects, it may be useful to add a render queuing option which allows the user to schedule the rendering of multiple scenes one after another in a user-specified order. This would allow the user to leave the program unattended while it renders and not have to worry about coming back to start each of the new renders separately.

Textures

There are 4 primary ways which rays will be altered when interacting with objects of different materials. For opaque materials, the light will be scattered either diffusely (where each ray is given a random angle in the opposite direction to the way it intersected with the object) to give a matte texture or specularly (where the angle of incidence is equal to the angle of reflection) for reflective surfaces. Transparent materials will allow most light to pass through them with potentially some distortion or they will reflect a small amount of light back like a reflective material would (this is to make the material still visible and not look like there is simply nothing there. For example, reflections off water will be slightly stretched parallel to the viewing direction and the intensity of the light will be slightly reduced (a similar effect will be seen on metallic materials, using a blur rather than a stretch). Finally, translucent materials will allow some of the light to pass through and some light to be diffusely or specularly scattered spending on the material. One method of doing this is to have some of the light reflected off the surface and then the rest of the light diffusely scattered after they have partially passed through the surface of the material - however this method is somewhat complicated to calculate so will only be added in a later sprint if I have a successful translucency implementation already as well as spare time left over. The only other way which materials will be rendered without the use of a texture map is using random perlin noise to generate a non-repeating marble effect.

Texture mapping will be used to represent any other materials the user wishes to import and use in their scene. This, along with the generated textures allows the user to create more realistic objects and scenes as well as making the scene more interesting to look at which is important for making visual products such as games and movies more engaging for the consumer.

I do not expect the bulk of this feature to take long to implement and it is a core aspect of rendering interesting scenes so I plan to complete this during one of my earlier sprints.

Effects

Similar to textures, effects such as cinematics, volumes (e.g. fog, smoke) and motion blur make the scene more engaging to look at and can contribute greatly to the feel and theme of the image. Additionally, effects can help with the realism of a scene - making it appear more professional. These effects are as follows:

- ➔ Ambient Occlusion
  - ◆ Makes lighting and shadows smoother and more realistic
- ➔ Tyndall Effect
  - ◆ More realistic interaction between a liquid and the light passing through it
- ➔ Luminance
  - ◆ Secondary light sources

- ➔ Defocus Blur
    - Blurring parts of the image outside the depth of field to render scenes more realistically as if it were taken by a real camera.

Some of these effects are more complicated than others but many are definitely achievable so I plan to implement the majority of these after I have implemented the texture functionality.

Path Tracing

Sphere intersection and surface normals will be required to create my first ray traced images, later down the line I will implement quadrilateral intersection in order to work with more complex scenes. After this I will implement bounding volume hierarchies to support more objects in a single scene without impacting rendering times too much. This provides support for users or teams working on larger scale professional projects without hitting rendering bottlenecks - however is not really necessary for smaller scale personal projects as I am only rendering still images rather than video which does not take nearly as much time. Finally, I will implement antialiasing where an average is taken from multiple samples around each pixel in order to render smoother curves for less pixelated images.

The majority of these should not take a very long time to implement and are vital to creating a fully functioning ray tracer, so I plan to complete most of these in early stages of development and then further optimise the BVH if I have the time capacity for it later on.

Version Control

Upon opening the program, the user will have to login to their account in order to access their projects - these accounts will only be saved locally with the purpose of allowing multiple users to use the program on the same device. When a user saves a scene (the data will be saved within their account only) they will be given the option to either overwrite the currently saved scene data or create a new version copy of the project, as well as being able to add a comment when they save to indicate the changes made at this point in their development - these comments along with a date/timestamp will be stored in a metadata file within the project folder. This allows users to decide how rigorously they wish to keep track of the project's version control depending on the scale of the project and how many people are working on it. Meaning projects of varying scale from different types of developers can be carried out using my software.

When opening a project, the versions will be displayed in reverse chronological order (with the most recent version at the top). This makes it easy to keep track of projects with lots of different versions from different points in time.

Additionally, I wish to implement a system which allows users to create version branches in their project folders which they can create separate versions of the project within. Users can then merge different versions back into a single branch. This merging mechanic may also become cross-user if I have the time but is not integral to the use of the software as people who wish to work collaboratively could just share a team account.

The folder hierarchy of this system is as follows:

```
App/
└── User/
        ├── preferences/
        ├── assets/
        │       ├── textures/
        │       └── backgrounds/
        └── projects/
                └── project name/
                        └── branch/
                                ├── version name/
                                │       ├── scene data/
                                │       ├── metadata/
                                │       └── unique assets/
                                └── common assets/
```

I do not believe that any of this (apart from potentially the merging feature) will take very long to implement as it is mostly path and file handling which I am fairly confident in already. However, much of this is not required for the function of the program as a whole and simply just improves the user experience, particularly for larger projects and teams. Because of this, I will most likely complete this feature during a later stage of the project development process.

Limitations

My software will be made for rendering still images which means I will not be implementing support for videos or real-time rendering. These may be largely impactful additions but are primarily used for larger scale professional productions and require a lot of computational power to perform - as well as being extremely difficult to develop. These criteria do not fit the majority of my target demographic which is solo developers and small teams.

Additionally, I will not be providing support for cross-device user interaction as it would require connecting to the internet in order to login which may be complicated to set up and takes more development time away from the ray tracing itself. This means that users will not be able to edit the same project at the same time - however this, again, is usually only something needed for big teams working on larger scale projects.

Requirements

Non-Functional Requirements

Development

For the ray tracing portion of my program I was considering using either rust or c++ as my programming language due to their speed and popularity for graphics and simulation - which means there are lots of resources available to learn from when I run into issues during development. Rust is a newer language so

does not have as many resources but I decided it would be preferable due to its safer memory management. I will not be using any external libraries or tools for this part of the project.

For the user interface I will be using Python with Tkinter as the UI interacts with the version control system directly - which is something I will also be developing in Python due to my confidence in the language and its built-in module support for file and path management. I chose Tkinter as my UI library and PyThreeJS for the 3D environment, both for similar reasons as my previous choices (popularity for use cases similar to my own and prior experience).

Additionally, I will be using a combination of Python and SQL in order to manage project and user data within my app. This is because I will already be using python for a large portion of my code, so it reduces the complexity of making different aspects of my code base interact, and it is easy to integrate with SQL.

I will require a computer with a graphics card in order to test my code during development due to the interaction of PyThreeJS and my rust code with the GPU.

My project has a number of different parts which depend on each other, and the potential for requirements to change throughout development. Because of this complexity I have decided to use the agile software development lifecycle, which allows me to build a basic shell for the software and iteratively build on it in layers - ensuring that all parts of the software are up to date with each other. Additionally, my stakeholders are far more experienced in 3D graphical development than myself so using a methodology with frequent user feedback helps me to handle changing requirements and ensure that my final product meets the needs of my target market - making use of quality assurance testing after each sprint.

Usage

Since PyThreeJS and my rust ray tracing program will both make use of the GPU in order to render images, the user would require a graphics card to run my software. This limits the number of devices my program can run on but is necessary to render images of high quality in realistic amounts of time because of the GPUs parallel processing capabilities.

Functional Requirements

| Requirement | Priority | Raised By | Description | Feature |

Object Manager	Must Have	Me (Jamie)	Tab with visual list of all the objects in the scene and their variable attributes on display to edit, as well as buttons to add and remove objects from the scene	Scene Creator
Positionable Camera	Must Have	Me (Jamie)	Method for user to move the camera to different positions and orientations around a scene	Scene Creator
Sphere Intersection	Must Have	Me (Jamie)	Calculating ray intersections with spheres	Path Tracing
Surface Normals	Must Have	Me (Jamie)	Calculating vectors normal to surfaces at the point of ray intersection	Path Tracing
Antialiasing	Must Have	Me (Jamie)	Taking multiple samples of a pixel at different nearby points and calculating an average colour value	Path Tracing
Diffuse Materials	Must Have	Me (Jamie)	Scattering rays randomly off objects which don't emit light in order to create a matte texture	Textures
Reflections	Must Have	Me (Jamie)	Specular scattering off light off an object	Textures
Dielectrics	Must Have	Me (Jamie)	Splitting single rays into separate reflected and refracted rays upon intersection to create transparent surfaces	Textures
Quadrilaterals	Must Have	Me (Jamie)	Calculating ray intersections with quadrilaterals	Path Tracing
Lights	Must Have	Me (Jamie)	Being able to include numerous light sources throughout the scene	Effects
Multiple Users	Must Have	Martin	Ability to sign in with different users who are working on different scenes	Version Control
Savable	Must	Me	Allowing the user to save all the data for a scene (as opposed to just the final	Version

Scenes	Have	(Jamie)	render)	Control
Object Addition Menu	Must Have	Me (Jamie)	Pop up window when adding objects to choose their attributes, the quantity of the object you are adding and the position(s) of the object(s)	Scene Creator
Resolution Dropdown	Must Have	Me (Jamie)	Dropdown to edit the resolution of the rendered image (from a range of standard options) without changing the aspect ratio	Settings Menu
Aspect Ratio Dropdown	Must Have	Jason	Dropdown to select aspect ratio of rendered image, from a range of standard options (including an option for a custom aspect ratio)	Settings Menu
Water Reflections	Must Have	Me (Jamie)	Blurring and stretching the reflections off water as well as reducing the intensity of light reflected in order to better replicate the way light bounces of the surface of water	Textures
Backgrounds	Must Have	Martin	Having a range of different types of backgrounds which can be applied to a scene for different effects and use cases	Scene Creator
Imports	Must Have	Martin	An option to import additional textures and backgrounds from your computer file system	Scene Creator
Import Validity	Must Have	Martin, Me (Jamie)	Checks to ensure imported texture and background files are the correct format	Scene Creator
Defocus Blur	Should Have	Me (Jamie)	Scaling blurring at different depths to recreate the "depth of field" effect caused by real cameras because they gather light through a large hole rather than a point	Effects
Volumes	Should Have	Me (Jamie)	Creating the effect of translucent gases such as smoke or fog within a scene	Effects
Texture Mapping	Should Have	Martin	Applying material effects to objects in the scene	Textures
Perlin Noise	Should Have	Me (Jamie)	Creating non-repeating textures such as marble using random noise generation	Textures
Instances	Should	Me	Object transformations such as rotations and translations	Scene

	Have	(Jamie)		Creator
Texture/Background Manager	Should Have	Jason	Galleries showing all of your downloaded textures/backgrounds. Glass, steel & marble will be default materials available which are generated without using a texture map	Settings Menu
Performance Toggles	Should Have	Me (Jamie)	Toggles to include or disclude effects such as shadows, metallic reflections, reflections, volumes, motion blur	Settings Menu
Metallic Blurring	Should Have	Me (Jamie)	Adding a smooth blur to the reflection effect seen on metallic objects to make it appear more natural and realistic	Textures
Ambient Occlusion	Should Have	Me (Jamie)	Calculating a scalar value for each point on a surface to determine the amount of ambient light being reflected off it from the rest of the environment. This helps to smooth shadows, making 3D objects appear more realistic	Effects
Renders Slider	Should Have	Jason	Creating sliders to edit preferences such as depth of field, when rendering so values can be adjusted more intuitively	Settings Menu
Saving Options	Should Have	Jason	Allow users to choose between overwriting the current file and creating a new file with the updated content when saving projects	Version Control
Version Descriptions	Should Have	Me (Jamie), Martin	Allowing the user to add a short comment or description to each version of a project as they save it - similar to Git's commit messages	Version Control
Gallery Searching	Should Have	Jason	Assigning multiple search terms to each texture/background to make searching more intuitive	Settings Menu
Dark Mode	Should Have	Martin	Option to view the entire interface in a dark mode	Scene Creator
Chronological Versions	Should Have	Me (Jamie), Martin	Showing the dates on which versions of a projet were saved and displaying them in reverse chronological order (top version most recent)	Version Control
Scene Templates	Could Have	Me (Jamie)	Introducing a few basic preset scenes for common use cases	Scene Creator

Cinematic Camera Effects	Could Have	Me (Jamie)	Effects such as bloom, lens flares, chromatic aberration & vignette	Effects
Visual Scene Previews	Could Have	Me (Jamie)	Low detail preview of the scene as it is being created and edited	Scene Creator
Illustrated Dropdowns	Could Have	Me (Jamie), Martin	Small diagrams next to dropdown option descriptions to illustrate the function of the option	Scene Creator
Bounding Volume Hierarchies	Could Have	Me (Jamie)	Discarding subsets of ray calculations depending on intersections with larger bounding volumes	Path Tracing
Subsurface Scattering	Could Have	Me (Jamie)	Rendering realistic translucent materials by reflecting some rays off the surface and scattering the rest after they have partially passed through the object	Textures
Merging Scenes	Could Have	Me (Jamie)	Merging multiple different scenes from the same user or across users	Version Control
Tyndall Effect	Could Have	Me (Jamie)	Increasing the visibility of light as it passes through certain fluids	Effects
Motion Blur	Could Have	Me (Jamie)	Creating the effect of movement in a still frame by adding blur to an object in one direction	Effects
Rendering Queue	Could Have	Me (Jamie)	If you want to render multiple scenes (or the same scene in various different ways), which might take some time you can create a queue of scenes to be rendered which will start one after another automatically	Scene Creator
Parallel Rendering	Could Have	Me (Jamie)	If you want to render multiple scenes or versions of a scene at once, you can select a number of scenes to render in parallel with each other	Scene Creator
Project Branching	Could Have	Me (Jamie), Martin	Allowing users to make numerous version branches from different saves along the main branch	Version Control

Cross-device user interaction	Won't Have	Me (Jamie)	Users from different devices collaborating on one project, making use of the internet	Version Control
Video Rendering	Won't Have	Me (Jamie)	The user will not be able to string together multiple versions of a scene and turn it into an animation	Scene Creator
Real Time Rendering	Won't Have	Me (Jamie)	The scene won't be rendered in real time using ray tracing as it is being edited	Scene Creator

High Level Success Criteria

1. **Settings & main menu** should be intuitive and easy to navigate
    - This is to ensure that users who are new to 3D development are still able to make use of my software and can gradually learn more advanced features

2. **Gallery** should support storing, displaying and searching plenty (>= 100) of textures/backgrounds
    - In order to support larger scale projects, as well as users who have many projects built up - it is important that frequently used assets are readily available without the user having to waste time re-importing them

3. **Rendered images** should complete in a short amount of time and be produced to accurately represent the scene
    - I have to compete with other currently available tools in order to appeal to my target market which means my software needs to produce high quality images without sacrificing time

4. Software should support **multiple user accounts** with different projects in each account and be able to quickly switch between accounts and projects
    - Many inexperienced groups of users may only have access to one machine that meets the hardware requirements to run my software, so having multiple user accounts on one device is essential to delivering my product to as many people as possible

5. **3D environment** should be visually appealing and have enough contrast to clearly see different aspects of the scene
    - The 3D environment is most likely going to be the most important feature in terms of user experience when using my software. So, especially when dealing with complex scenes, it is very important that the user can easily see what they are working on without getting lost

6. Users should be able to well manage **different versions** of a project and smoothly navigate between branches without data getting lost or corrupted
    - 3D graphical design can be a very time consuming process especially when creating complicated projects. This means it is vital for the software to be extremely safe when dealing with project data - as losing it could result in many wasted user hours

7. Scenes should support the addition and editing of **many objects** without causing excessive lagging
    - This is primarily for supporting semi-professional projects which are likely to be of a larger scale and will have to deal with the addition and editing of a wide range of different objects throughout the scene

8. **Materials** and **generated textures** should be precisely mapped to shapes/objects and accurately represent the surface
    - Inconsistent texture mappings are quite common in low quality 3D graphics and decrease the realism of the scene drastically. This means it is important for my software to display textures accurately if I want to be able to render competitive realistic scenes

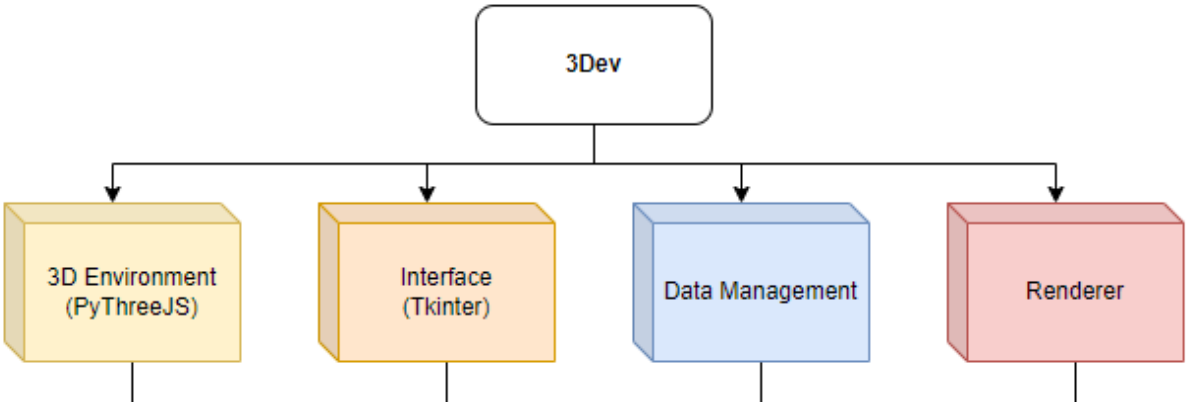
9. Users should be able to schedule and run **multiple renders** at once without having to manually start each render one after the other
    - Users may need to keep track of many different aspects of their projects e.g. in game development, which don't involve my software so this removes unnecessary time they have to spend on tasks which could be automated for them

10. **Importing/exporting** multiple assets/renders should be a swift and intuitive process which does not require unnecessary repetitive actions
    - For large scale projects, it is likely that a wide range of assets will have to be imported. Making this process as streamlined as possible gives users more time to spend actually developing their scene

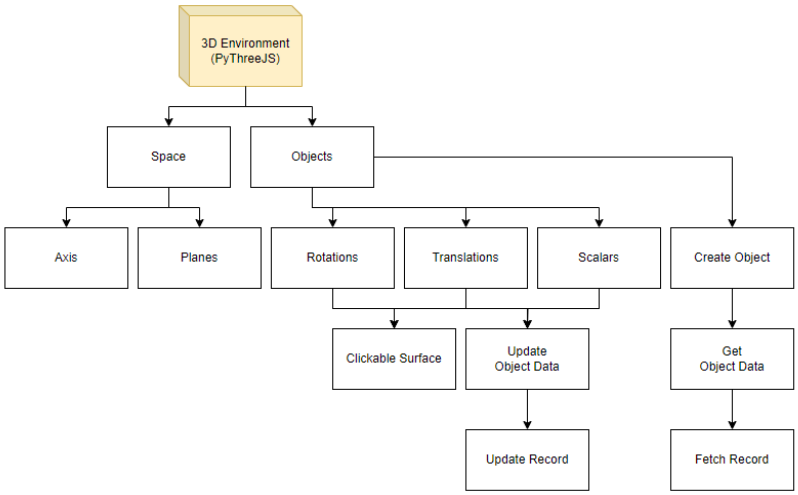
Design

Functional Decomposition

There are four main sections of development which my project will be split into - these were mentioned in the non-functional requirements section. The user interface (using Tkinter), 3D environment (with PyThreeJS) and Data Management will be coded in Python while the rendering portion of the program will be made in Rust.

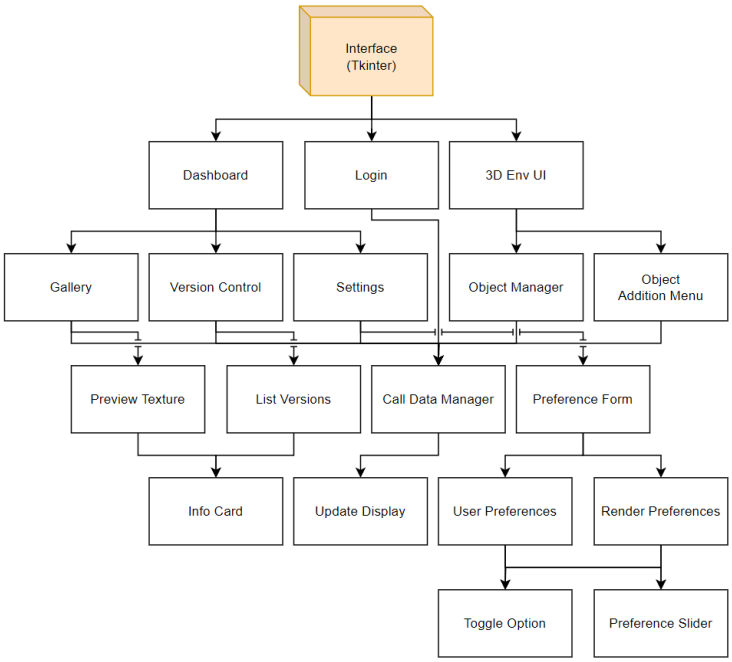


3D Environment



The implementation of the 3D environment, specifically the space portion of it, will be covered mostly under the PyThreeJS library. In terms of objects, there is some shared functionality which can be seen in the diagram above. This mostly involves reacting to how users interact with the objects in their scene and updating the object data accordingly. This means that I am likely going to need a table to store object data as the scene changes and objects are added/edited. I will decompose the data-management section of my project in order to get a better understanding for how this object data will be managed and affect other areas of the project.

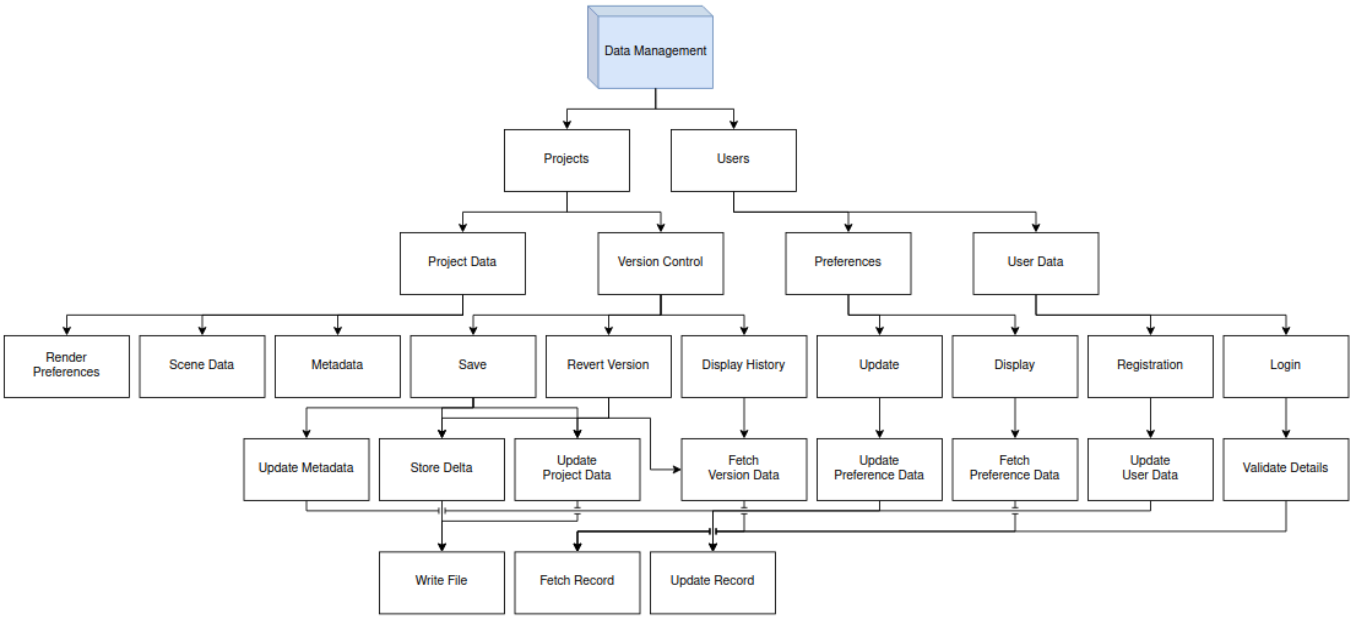
Interface



Almost all elements of my interface will interact with the data manager as the majority of the interactions the user will have with the software impact either project or user data, which means database tables need to be updated. Different menus such as user/preference forms use the same elements for representing different pieces of information - this indicates the need for a general UI module which I can call from in different areas of the program to reduce redundant code.

Decomposing the different windows I will have into their UI components allows me to get an idea for which aspects of the interface can be made modular for other functionality and which ones are more niche. I will use this distinction to prioritise working on more widely used elements in earlier sprints. Additionally, I can see that the 3D environment is going to be one of the first interface sections I will have to tackle as many of the other preferences and menus can only be properly tested based on their effect on it.

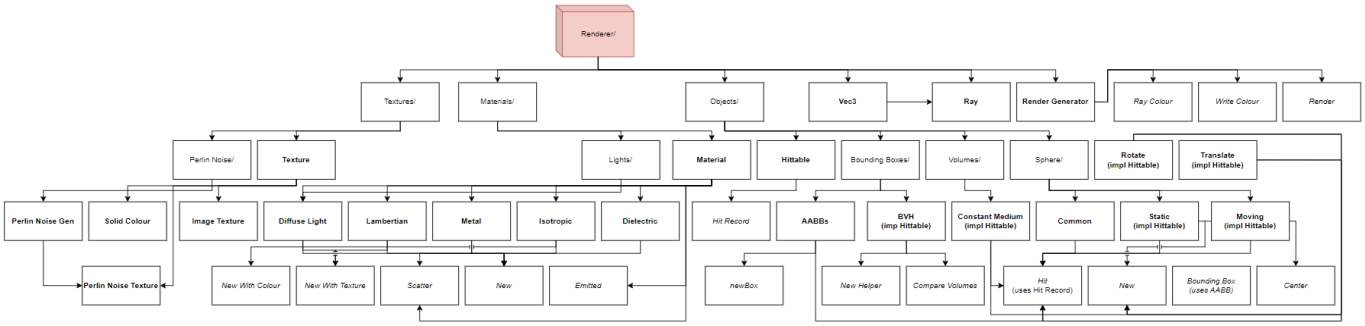
Data Management



Data management will fall under user data and project data. However, as seen in the diagram above, much of the management for project and user data involves the same subroutines. I can abstract these to make my data manager more adaptable to new tables.

Breaking down all the data I am handling into these sections gives me an idea for what tables I will need to construct my database. Viewing the connections between different branches also shows me what kind of communication will be needed between the different tables. I will use this in my sprint design to create entity-relationship diagrams for my database and also design each of my tables.

Renderer



Most of the renderer portion of my software has been broken down under either objects, materials or textures (these will have their own folders - denoted by the "/"). Within each of these folders there will be a number of sub-folders and modules (titled in bold) for different behaviours. In many cases, there are different forms of a particular behaviour which need to be implemented for different scenarios. Because of this, lots of the modules in each folder will dictate a behaviour which is a composite of the code in that file and behaviour from a higher level module. This relationship, as well as the folder hierarchy of the codebase, is indicated using the arrows. Finally, some behaviours will share different variations of the same method e.g. The different material types will all require a "*New*" method, to create a new instance of that material, specific to themselves. All methods have been indicated in my diagram using italics.

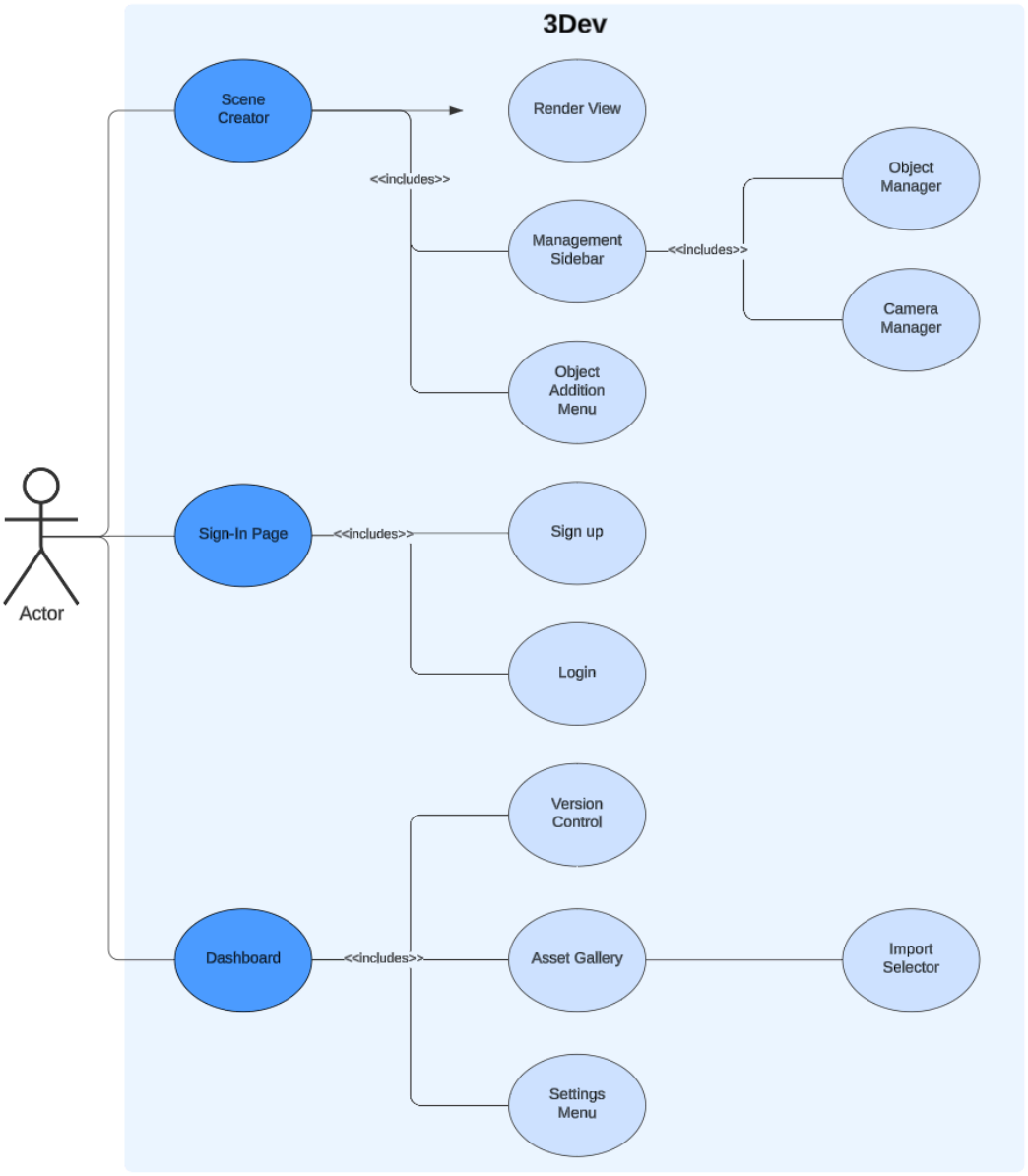
Outside of these 3 categories, I will have standalone modules for the vector calculations of individual rays, generating rendered images using all the other modules, and a vector class to help with ray calculations.

This decomposition has made it clear that the majority of my work on the rendering portion of my software will be focused on dealing with objects. This is because of the larger number of sub-branches underneath the objects section of my diagram - which shows the number of modules/sub-folders and their respective complexity (how many different methods which go into making them). Because of this, and the fact that material and texture behaviours are reliant on objects to be applied to, it is important that I prioritise getting the objects implemented quickly once I begin development.

Technical Diagrams

I have decided to put the large majority of my project designs in one section before I begin development, rather than completing them sprint by sprint. This is because many aspects of my software, which are going to be covered over different sprints, are heavily reliant on each other - so it seems more reasonable for me to complete their designs at once. Designing in this way will avoid me having to inelegantly combine chunks of incomplete feature designs later on.

Use Case Diagram



3Dev will have 3 primary interfaces for the user to interact with - the scene creator, sign-in page and dashboard. I will now discuss the aspects of these interfaces that are going to be developed during sprint 1.

The **scene creator** is going to include the management sidebar as well as the object addition menu.

The **management sidebar** will have a section for **objects** and a section for the **camera**, both of which will contain various sliders, toggles and fields for the user to interact with.

The **object addition menu** will occupy the centre of the screen as a pop-up window when the user clicks on the object addition button, the user will be prompted to fill in fields regarding the attributes of the object they wish to add before saving and adding the object into the scene.

I will also be working on the **renderer** during this sprint which is going to be an extension of the scene creator interface as a completely separate window which opens once the user clicks the “Render” button for their scene. This has not been connected to the scene creator for this use case diagram as I will be developing the renderer completely separately from the rest of the app until the later stages of sprint two.

The **sign-in page** has two sections, **sign up** and **login**, which both require the user to fill in a username and password field. However, there are two password fields in the sign up section in order to confirm their choice, this is important because cloud or centralised data management was not within the scope of this project - meaning there is no two factor authentication as a backup in case the user unknowingly mistypes their password when signing up.

The **dashboard** has 3 sub-sections within it - version control, asset gallery and settings. These key headings were chosen as they were outlined in earlier sections as primary features for the user to interact with.

Version control will contain a list of cards which each present one of the user's projects with a title, version number, creation date and preview image - as well as a launch button to open up the scene. At the top of the page there will be a section dedicated to starting a new project.

The **asset gallery** is going to have a similar layout, however, the “New Project” section will become an “Import Asset” section with a button to browse for files and an area to drag and drop them. Also, cards will be split into two sections for textures and backgrounds and laid out into more of a grid. Asset cards will contain a title, file size, import date and preview image.

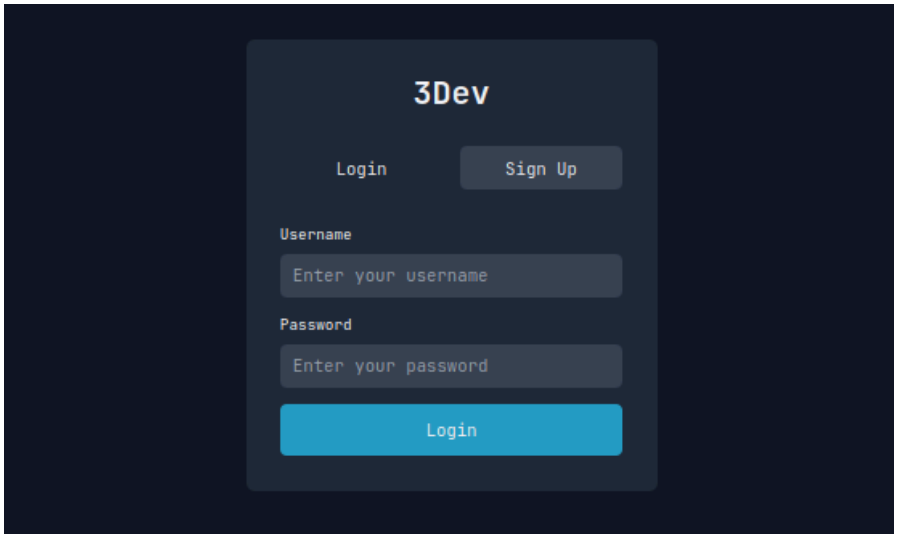
Settings will be split into sections, such as user preferences and appearance, for the user to scroll through - each section will contain a variety of fields, toggles and sliders. Finally, there will be a search bar at the top for finding various assets.

All of these interface designs will be justified under their respective sections found below. I am going to create wireframes to display how I will design each of these interfaces and their sub-sections. Additionally, flowcharts and various database diagrams will help me design the logic needed to manage the information around these interfaces.

Sign-In Page

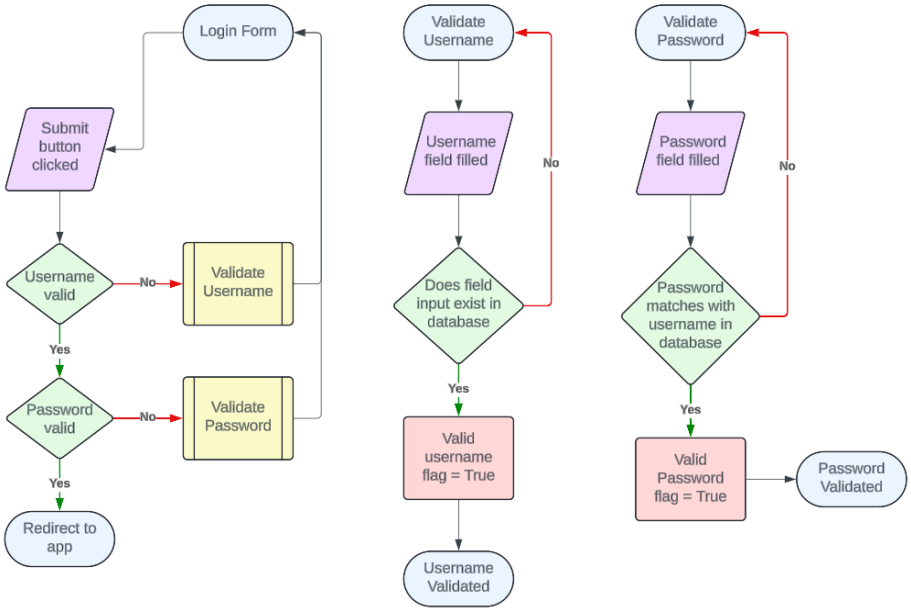
From this section on, I will begin designing the wireframes for different pages of my user interface. I have decided to put particular care into designing these wireframes effectively, rather than just simple layout concepts. I have made this decision based on the fact that python is not used particularly often for UI development so can be more difficult to deal with than others. Subsequently, I may find difficulty trying to experiment with designs during development if I do not have a specific image to emulate.

Login



This wireframe displays my design for the login page, which simply contains username and password fields as well as a “Login” button highlighted in blue. I plan to keep this blue theme throughout the design of my interfaces. Once implemented in code, the login button will change to a slightly darker blue whilst being hovered over and, if I have time in this sprint, will become momentarily smaller when clicked to indicate to the user that the click has been registered. The “Login” and “Sign Up” options are also highlighted differently to indicate which option is currently selected, however I think this current design is unclear as the Login option is the same colour as the background. This may make some users think that they are on the sign up page rather than the login page.

Aside from design, I am also going to need logic to process these interface inputs.



This flowchart is split into 3 sections, the main body of the program and 2 subroutines:

- **Main loop:** This piece of code will constantly be checking if the submit button has been clicked. Once it has, the “Validate Username” and “Validate Password” functions will be called - to check

these credentials. These functions return a boolean value to indicate if the credential is valid. If both of these pass True then the user is redirected into the app.

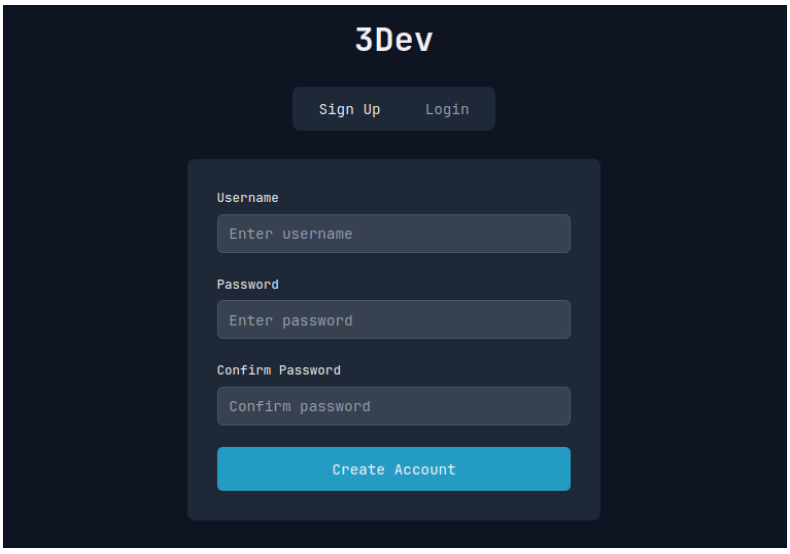
- ➔ **Validate Username:** Once the username field has been filled, the program checks whether that username exists in the database or not. If it does then the function returns True and if it does not then the user is prompted to re-enter the username.
- ➔ **Validate Password:** The password validation works just like the username validation but the program checks to see if the password given matches the username.

This is the table of variables I would require to put this logic into code:

Variable Name	Description	Data Type	Validation Required	Justification
username	Stores the username entered by the user	String	- Must be filled (non-empty) - Must exist in the database	Ensures a user is attempting to log in with an actual, known account. Prevents empty submissions.
password	Stores the password entered by the user	String	- Must be filled (non-empty) - Must match the password associated with the entered username	Ensures secure login by validating credentials. Prevents access with incorrect or blank inputs.
is_username_valid	Flag indicating if the username validation passed	Boolean	- Set to True if username exists in database, else False	Helps in separating and managing validation logic step-by-step.
is_password_valid	Flag indicating if the password validation passed	Boolean	- Set to True if password matches the one stored for the entered	Allows for detailed feedback and control of

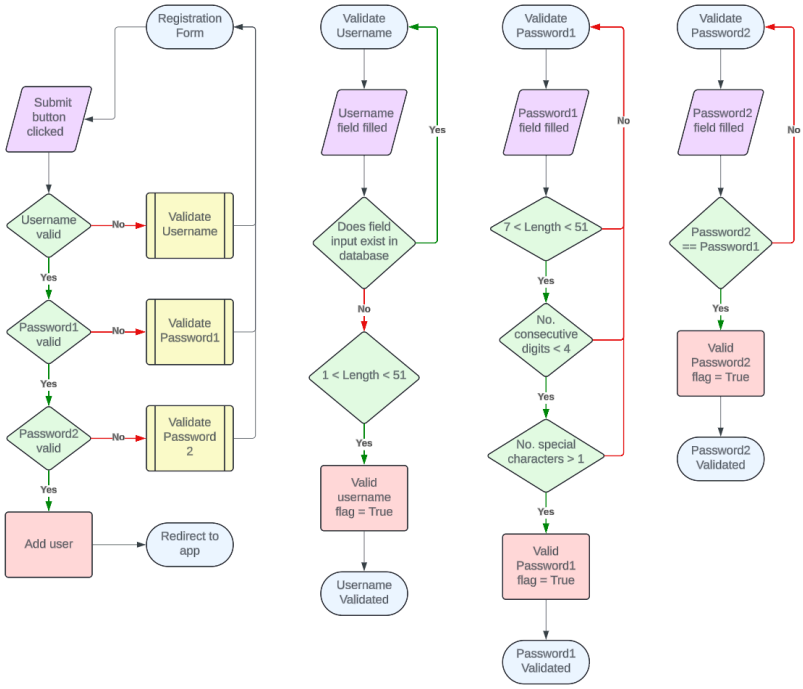
			username, else False	authentication flow.
user_authenticated	Flag to indicate if the user is successfully authenticated	Boolean	- Set to True only if both is_username_valid and is_password_valid are True	Ensures only verified users with valid credentials gain access.
submit_clicked	Flag to check if the submit button was clicked	Boolean	- Set to True when the submit action is triggered by the user	Allows the system to respond only to user-initiated authentication attempts.

Sign Up



Improving on my previous design, I separated the “Sign Up” and “Login” selection from the rest of the form and decided to opt for a white text highlight to indicate which option has been selected, which I think is far less ambiguous. Additionally, I moved the sign up option to the left and the login to the right as I believe this to be a more intuitive order - since the signing up is the first option that would be picked by a new user.

Finally, there is an additional password field on the sign up page for the user to confirm their choice.



The sign up page has similar logic to the login page with a few more checks:

- ➔ **Main Loop:** Once the submit button is clicked, the username, password1 and password2 flags are checked - calling each of their respective validation subroutines if false. If they are all true then a new entry is added to the database - using the information submitted - and the user is redirected to the dashboard.
- ➔ **Validate Username:** This function works exactly the same as the one from the login page but in reverse, so if the username does exist then the user is prompted to choose a new one. Otherwise, the username is valid as long as it's between 2 and 50 (inclusive) characters long.
- ➔ **Validate Password1:** This subroutine checks that the inputted password is at least 8 characters and at most 50 characters long, has no more than 3 consecutive digits and has at least 2 special characters.
- ➔ **Validate Password2:** Once password1 has been validated, this function checks that the second password field matches the first.

This is the table of variables I would require to put this logic into code:

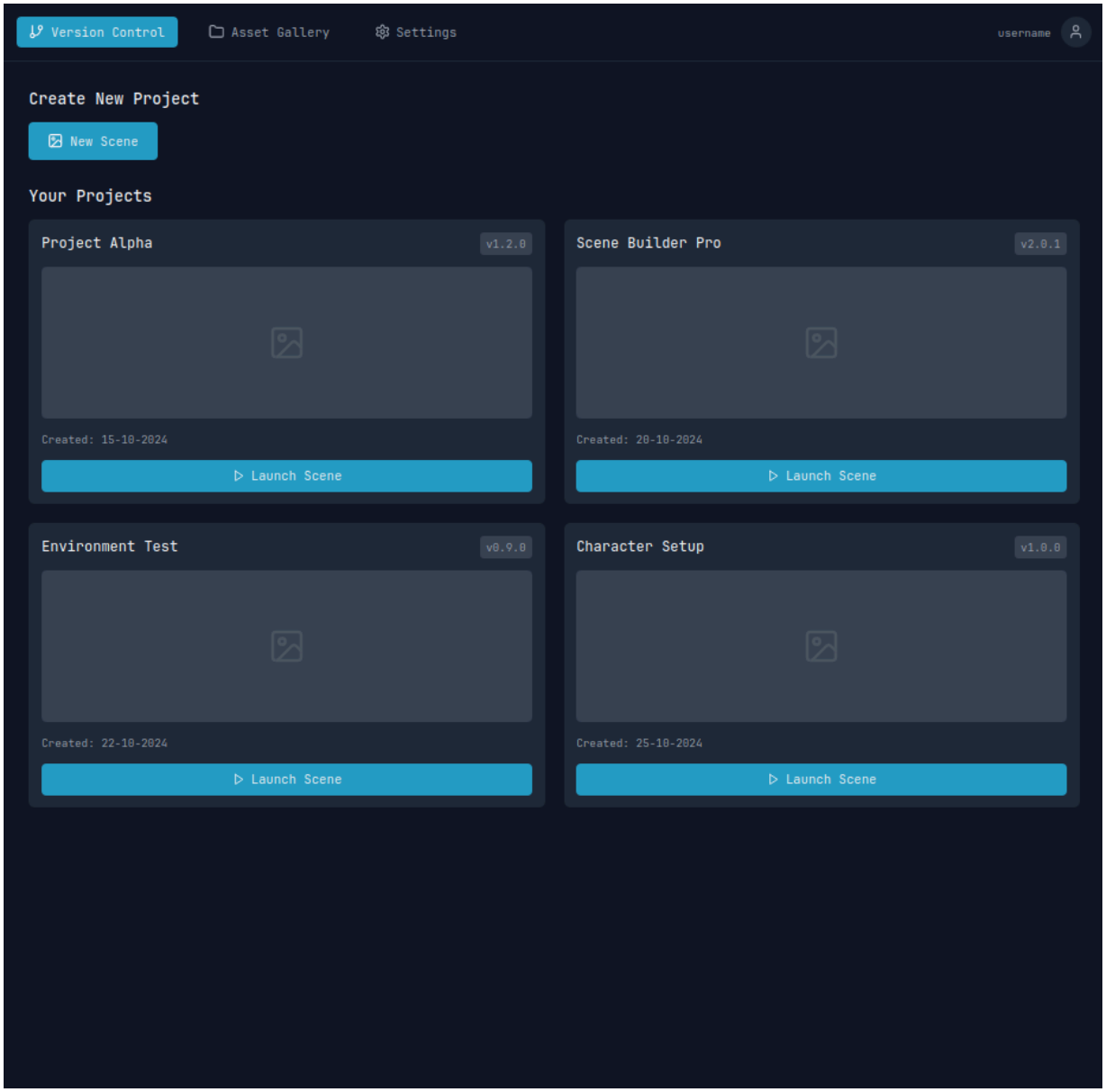
Variable Name	Description	Data Type	Validation Required	Justification

username	Stores the username entered by the user	String	- Must be filled (non-empty) - Length between 2 and 50 characters - Must not exist in database	Prevents duplicate accounts and enforces meaningful and manageable usernames.
password1	Stores the primary password entered by the user	String	- Must be filled (non-empty) - Length between 8 and 50 characters - No more than 3 consecutive digits - At least 1 special character	Enforces password strength for security and usability standards.
password2	Stores the confirmation password entered by the user	String	- Must be filled (non-empty) - Must match password1	Prevents typos during password creation and ensures the user knows their password.
is_username_valid	Flag indicating if the username validation passed	Boolean	- Set to True if username passes all validations, else False	Separates validation logic for better error handling and UI feedback.
is_password1_valid	Flag indicating if the primary password validation passed	Boolean	- Set to True if password1 meets all validation rules	Prevents weak or non-compliant passwords from being used.
is_password2_valid	Flag indicating if the confirmation	Boolean	- Set to True if password2	Ensures accuracy and avoids

	password validation passed		matches password1, else False	unintended password mismatch.
user_added	Flag to indicate if the user has been successfully added	Boolean	- Set to True only if all validations pass and the user is successfully created in the system	Ensures only validated and correctly entered data results in user account creation.
submit_clicked	Flag to check if the submit button was clicked	Boolean	- Set to True when the submit action is triggered by the user	Controls flow and prevents premature validation before user submission.

Dashboard

Version Control



In addition to the 3 section headers mentioned earlier, I have included a user icon and username in the top right corner which will remain present on all of the dashboard pages. The user can click on this to view their user data and/or return to the login page to access a different account without the need to restart the application. Avoiding the need for a restart will reduce the effort needed for users to switch between accounts and therefore encourage the use of the multi-user system.

The version control page, as well as the other dashboard pages to come, shares the same font and blue colour scheme used for the sign in page. This ensures design consistency throughout the application -

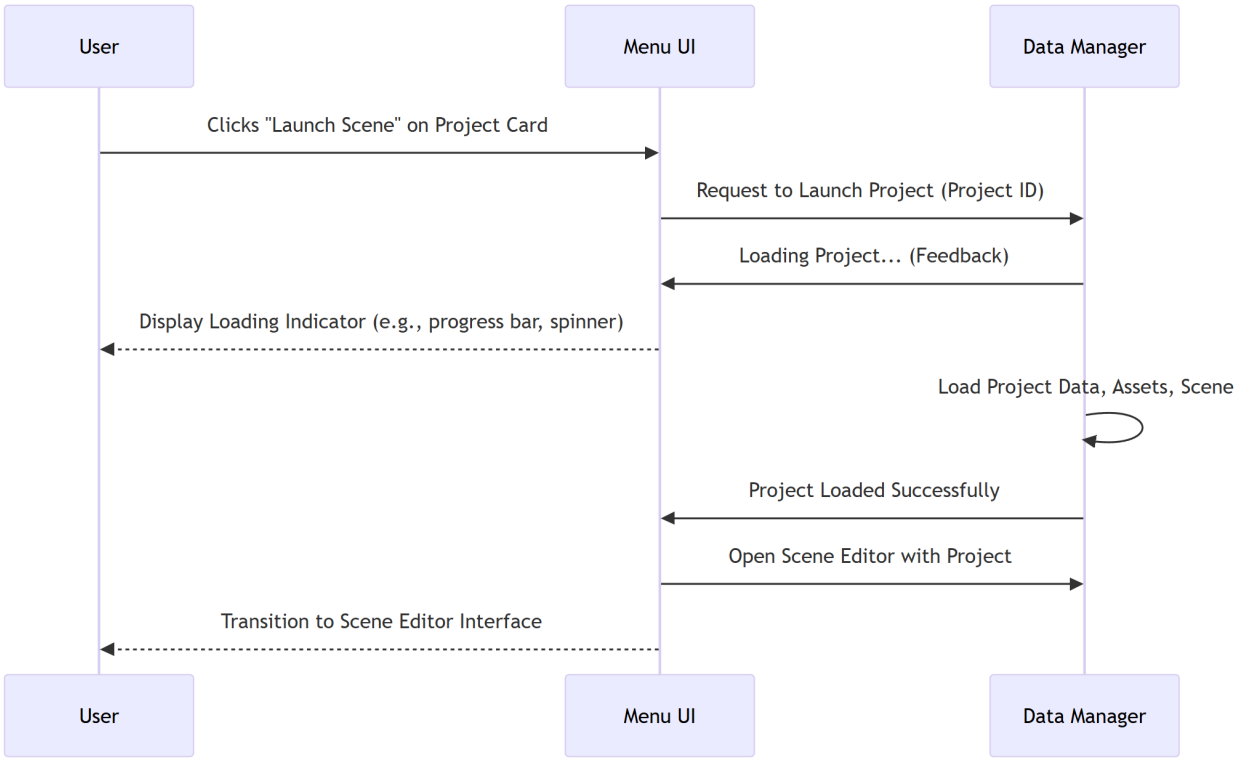
making elements predictable and shallowing the learning curve of using the app. All of these things reduce user errors and help to improve the overall aesthetics of the application.

Each project card is highlighted slightly lighter than the background colour to group the information around one project together. This makes it clear which pieces of information are in reference to which projects - avoiding user confusion as they have to navigate through their, potentially many, projects.

Because all of the dashboard menus are fairly complicated, in comparison to other interfaces such as the sign-in menu, I will create sequence diagrams which explain how user interactions with these interfaces communicate with the rest of the program.

Launch Scene

This is the most common action that will be taken by a user on this interface



1. **Initiation:** The User interacts with the Menu UI by clicking the "Launch Scene" button on a project card in the "Your Projects" section. This action signals the user's intention to open that project.

2. **Launch Request:** After receiving the "Launch Scene" click event, the Menu UI sends a request to the Data Manager . This request includes the information needed to identify the project being launched, likely through a Project ID.

3. **Initial Feedback:** After receiving the launch request, the Data Manager acknowledges it by sending a "Loading Project..." message back to the Menu UI. It is important to provide immediate feedback to the user that the system has recognised and begun processing their request. This is because fetching project data may take some time and the user may become impatient or frustrated without any kind of response.

4. **Loading Indicator:** The Menu UI translates the "Loading Project..." feedback into a visual cue for the User to see. This indicator will likely be a progress bar or spinner animation - these are dynamic visuals which prevent the impression that the application is frozen or unresponsive. This will help the user feel like progress is being made on their request and reduce potential frustration.

5. **Project Loading:** Simultaneously, the Data Manager works on retrieving all relevant information for the scene to be opened. These backend operations are abstracted from the user's view in this diagram. These operations will typically include:
    - Retrieving project data from database.
        - Loading associated assets (models, textures, etc.) which are required for the project.
        - Loading the scene data itself, which defines the 3D environment and objects within it.

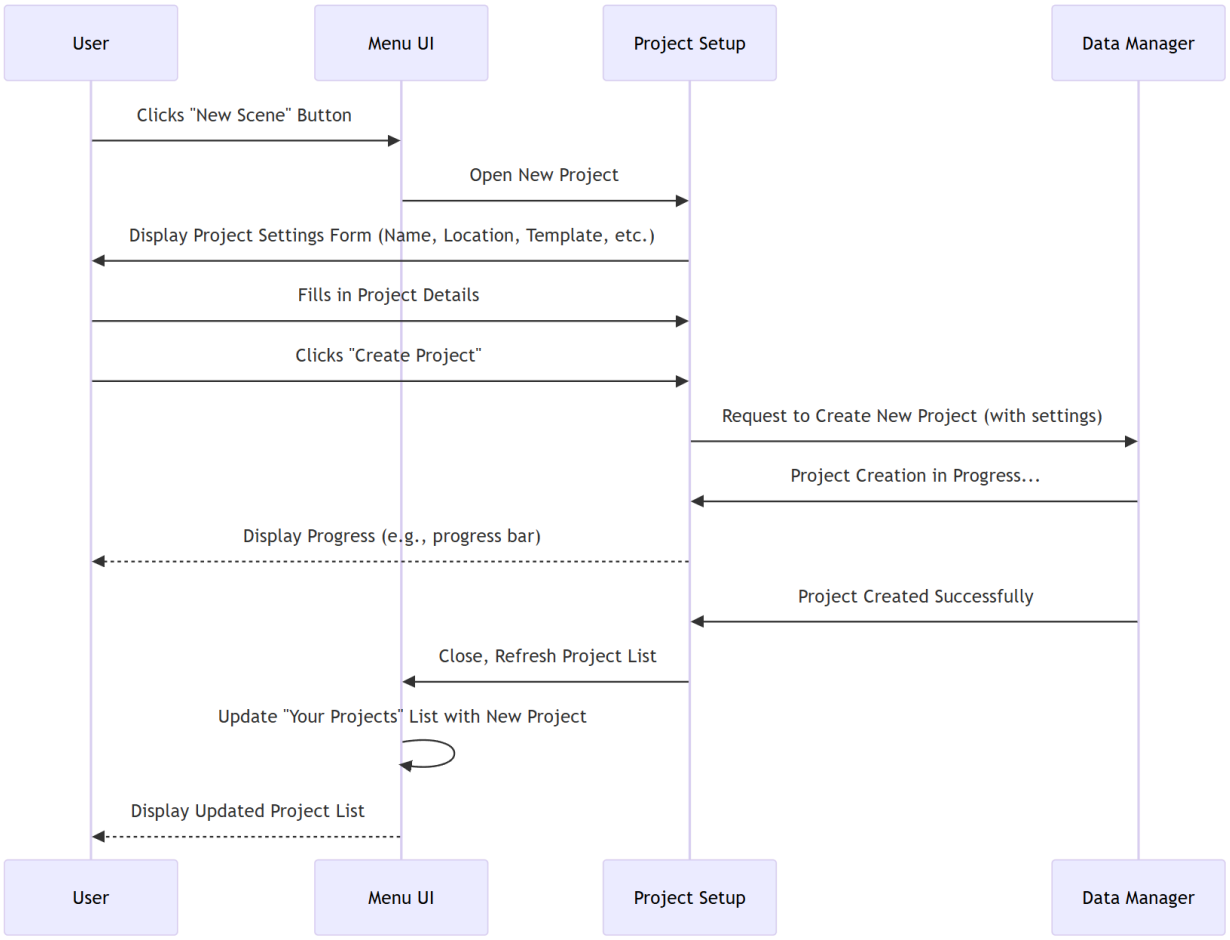
6. **Successful Load:** Once the Data Manager has successfully loaded the project data, it sends a "Project Loaded Successfully" message back to the Menu UI. This indicates that the backend is ready and the project is prepared for the user - an important step so that the software is actively communicating with the user and keeping them up to date on the status of their request.

7. **Open Scene Editor:** The Menu UI , upon receiving the success message, instructs the opening of the Scene Editor. This action initiates the transition from the dashboard interface to the primary 3D environment.

8. **Transition to Scene Editor:** Finally, the Menu UI transitions between the interfaces. It closes the project menu frame and displays the Scene Editor Interface to the User. This completes the project launch sequence so the user is now able to begin working on their project.

New Scene

The new scene action is less common than launch scene but requires a few extra steps to create a new project and then launch it as well



1. **Initiation:** This process starts once the User clicks the "New Scene" button within the Menu UI. This action signals the user's intention to create a scene.

2. **Open Project:** The Menu UI, responding to the "New Scene" click, triggers the opening of the Project Setup menu. This will be a separate UI component to guide the user through the configuration of a new project. Separating the “New Project” menu from the main Menu UI creates a visual distinction to show the user where to focus their attention. This prevents mistakes being made during the initialisation of a new project due to the user being otherwise occupied with different areas of the interface.

3. **Display “New Project” Form:** The Project Setup menu presents a form to the User. This form is designed to collect necessary information for creating a new project. This form will include fields for:
    - Project name
    - Project version number/name
    - Other relevant initial settings for the new project.

    Some information such as project creator and time of creation will be stored automatically - the full extent of project storage will be explained during the design of the Data Manager (later on in this section). This streamlines the project creation process for the user and eliminates any redundancy in

the information being filled out, which saves time for the user and improves the efficiency of their workflow.

4. **User Input:** The User fills in the required details on Project Settings Form, and customises project settings to their demands. The use of a form to make the user specify project details, before its creation, saves time since they do not have to manually navigate the project editor in order to make the required changes. This allows the user to start putting together their scene as soon as the editor is opened for them.

5. **Create Project Request:** Once the user has finished the form and clicks a "Create Project" or “Submit” button, the Project Setup menu sends a new project request to the Data Manager. This includes all the project settings and information gathered via the form.

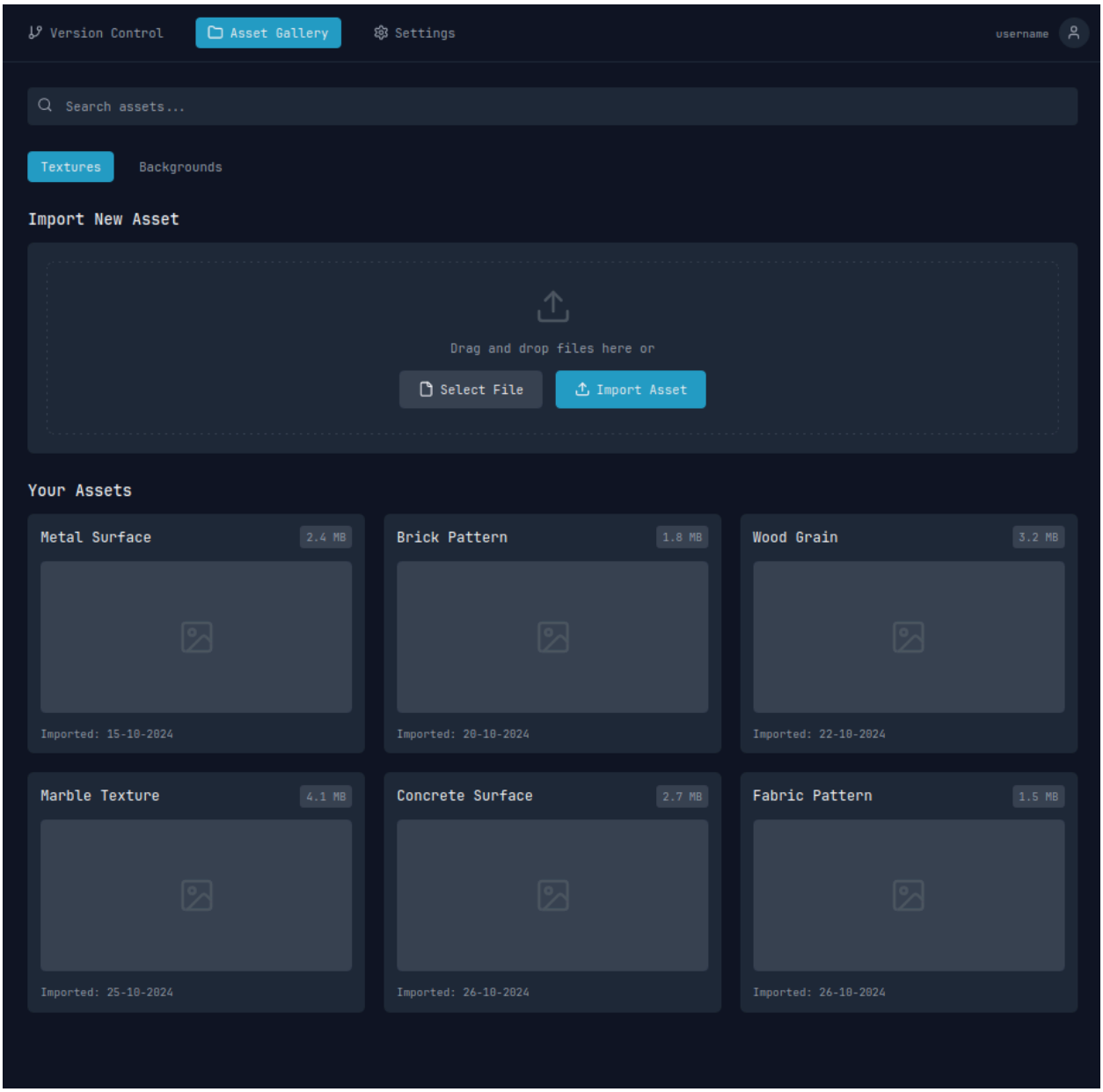
6. **Project Creation:** The Data Manager acknowledges this request and sends a "Project Creation in Progress..." message back to the Project Setup menu. This is, again, for providing dynamic feedback, showing that the project is now being created. This follows the same justification given for the feedback in the first diagram.

7. **Successful Creation:** After the Data Manager has successfully completed the creation process (involving the creation of project folders, initialisation of project files, setting up an empty scene, etc.), it sends a "Project Created Successfully" message back to the Project Setup menu. This is, again, providing the same feedback to the user which was justified for the first diagram

8. **Close Menu & Refresh List:** The Project Setup menu receives the success confirmation and the Menu UI closes it. The Menu UI also refreshes the "Your Projects" list. This ensures the newly created project will now be visible for the user in the project management menu so that they can load it again in the future. This involves fetching all the latest project data (including the new project) and re-displaying project cards.

9. **Launch New Project:** The same steps from the previous diagram are now completed to launch this new scene (if the user has specified in their settings that they want this action to be performed automatically upon the creation of a new project). This acts to avoid the redundant action of the user having to open new projects manually every time one is created.

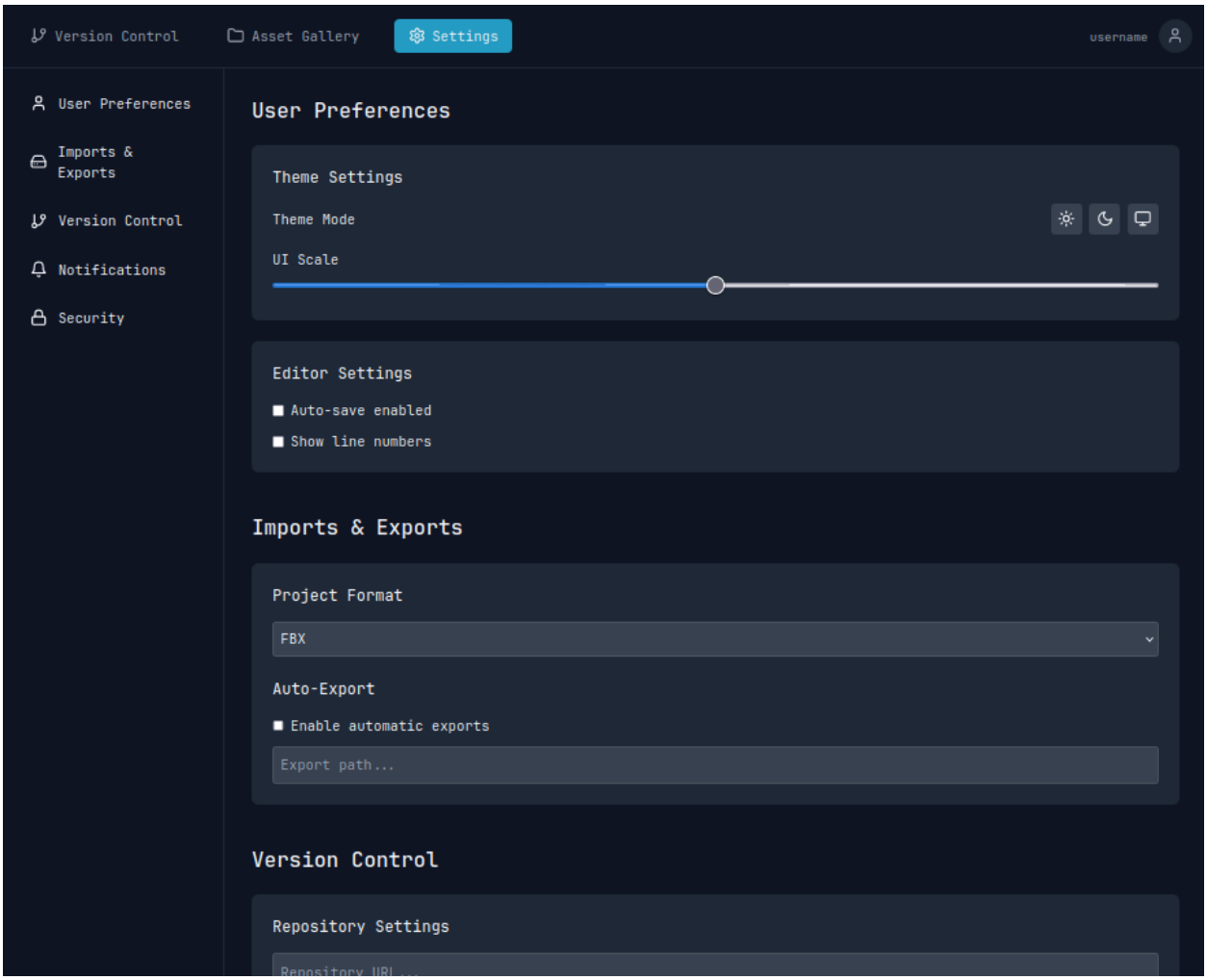
Asset Gallery



The asset gallery uses the same card format and reasoning as the version control page but, due to the likelihood of a user having substantially more assets than projects, in a 3 column grid format. This allows more cards to be displayed on screen at any one time, which reduces the time needed for a user to scroll through and search their assets.

The “textures”/”backgrounds” buttons have been placed above the “Import New Asset” section so that users do not have to specify which section they are importing to for every single asset, rather, they can remain on the textures section to import a number of textures in succession which will all be automatically allocated to the textures library. This saves time for the user and reduces repetition when importing multiple of one kind of asset.

Settings

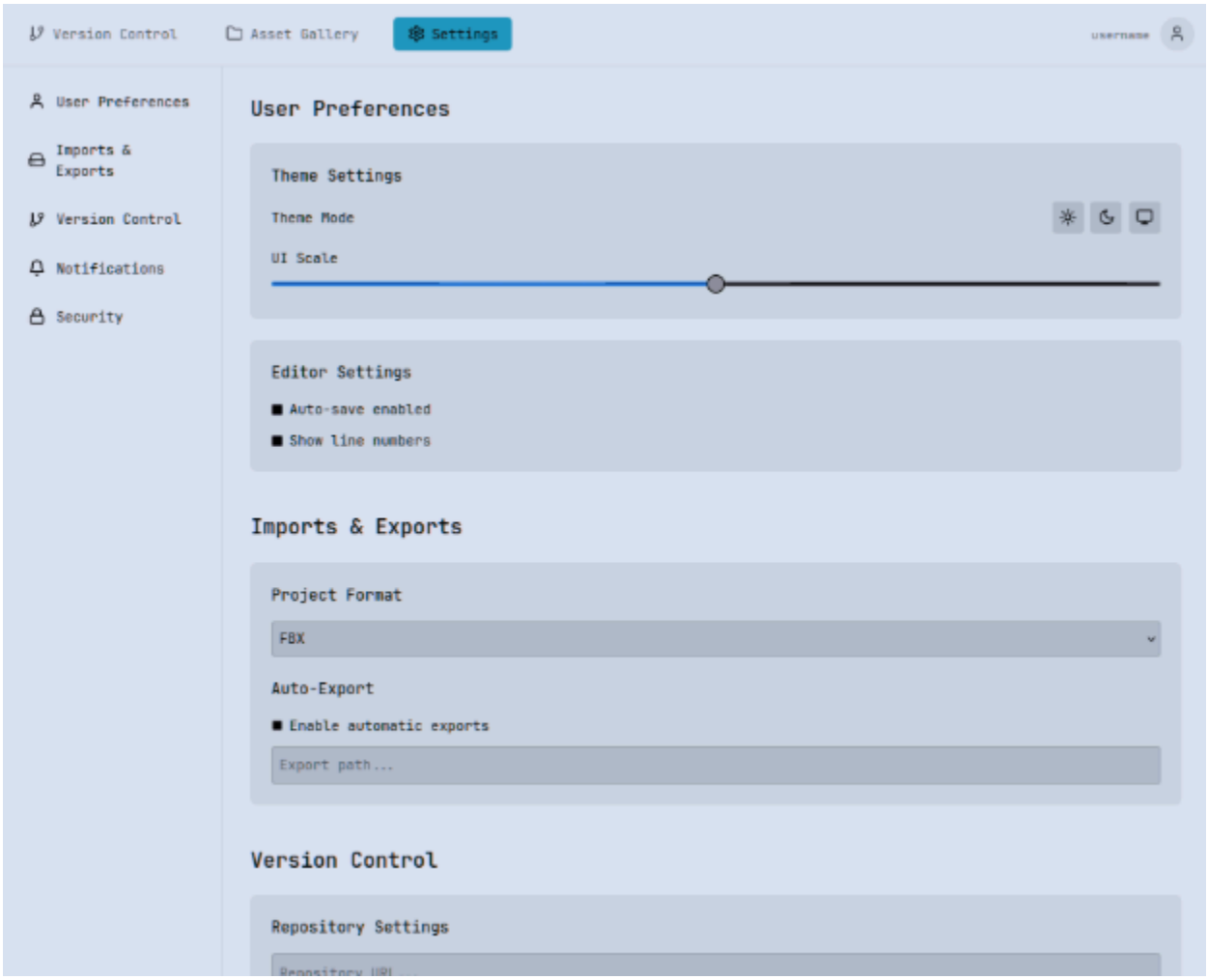


The settings menu is a continuous page that the user can scroll through to access all of the different sections, but there is also a sidebar with tabs for jumping between sections. This allows the user to decide how they prefer to navigate through their settings - streamlining their use of the menu and saving more time for actual project development.

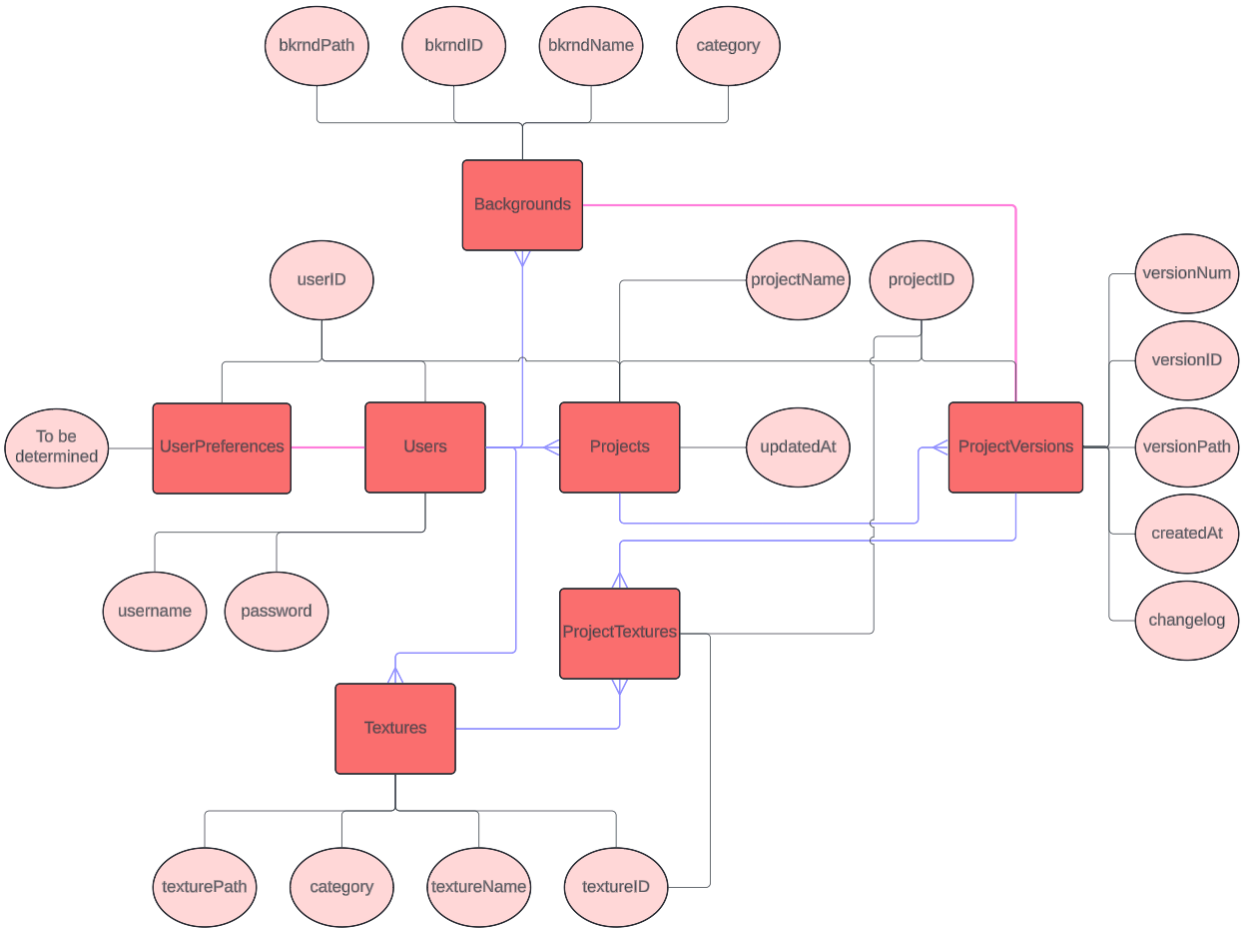
The options in this wireframe are mostly just for example of how the page will be laid out, similar to the sample projects and textures used for the other pages. Once I begin developing the code it will become more clear what preferences and options I will be able to offer to the user.

My original plan for the 3 sprints mentioned data management as its own chunk of development but because of the realisations made during sprint one (which lead to the implementation of the utility methods used throughout the code) I have decided to just develop the data management features as and when they are needed for the features I am developing. Since I am doing a lot of the UI to do with project and user management in this sprint, a lot of data management will be covered as well.

All of these interfaces will be made with their light mode counterparts, such as the following:



Data Management



This entity relationship diagram shows how the user and project management systems work together to manage data most efficiently. Each red box is a table and each red oval shows the fields related to it. Fields ending in “ID” are primary keys which are also used as foreign keys to connect two tables, this can be observed in this diagram when a field is connected to more than one table - such as *ProjectID* being connected to both *Projects* and *ProjectVersions*.

Pink lines between tables represent a one-to-one relationship while purple arrows represent a many-to-one relationship.

Starting with user data, I was considering keeping all core user data and user preference data in one table, since the two tables are going to be bijective anyway. However, because those sections of data are going to be used in separate contexts throughout my code, it makes it easier to keep them separate for the sake of simplifying my SQL code and making it more readable. I have not yet decided on the fields that will be included under user preference data, hence the “To be determined” on this diagram.

Asset data is split into two tables, textures and backgrounds, these tables are connected to both the user table and projects table. However, since projects can have more than one texture and any one texture can be used in more than one project, there is an additional *ProjectTextures* table in order to avoid a

many-to-many relationship. Backgrounds and textures have been separated into two different tables for similar reasons to the user data and user preferences - this simplifies the SQL when accessing the data because they are used in two different (however similar) situations.

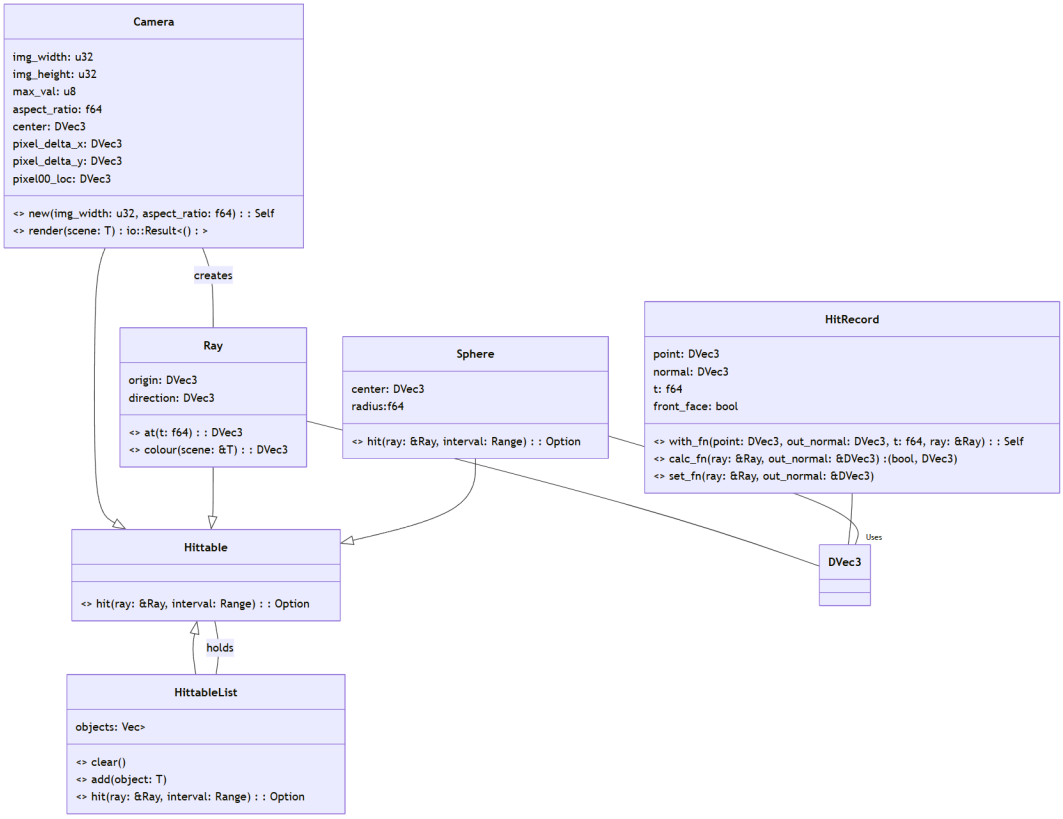
The *Projects* table links to *Users*, *Backgrounds*, *Textures* (via *ProjectTextures*) and *ProjectVersions*. This table stores the metadata for a project which is not version specific, when the user wants to access their project, a query will be made using the *ProjectID* to find all of the version records relating to that project and pull the path from the record with the highest version number (assuming the user is trying to access their most recent version).

Renderer

Class Diagram

The rendering portion of my code is going to be heavily class based as I am not going to have any main.rs file after the development is complete - I will have a group of modules (containing classes and methods) which will be called using the lib.rs file.

I am now going to create a class diagram for the early stages of my ray tracer which will be used to generate my first images and scenes. I am not going to create a class diagram of the more complicated features at this time since I am currently unsure of which features will be attainable and how I am going to have to implement them. If necessary, I will produce another class diagram at a later time which will be included in a further section of this document.



The camera class initialises most of the constants needed to set up the scene and is also responsible for rendering the final image. This render method makes use of the ray class as it sends them into the scene to collide with Hittable objects.

All objects (currently just the sphere class) in the scene are given the Hittable trait (traits in rust are similar to interfaces in other object oriented languages) which defines their shared behaviour of creating a HitRecord object if a ray intersects with them.

The HitRecord class is responsible for performing surface normal operations and storing all the other relevant data involved with ray-object intersections. This information is then used when calculating the colour of the ray.

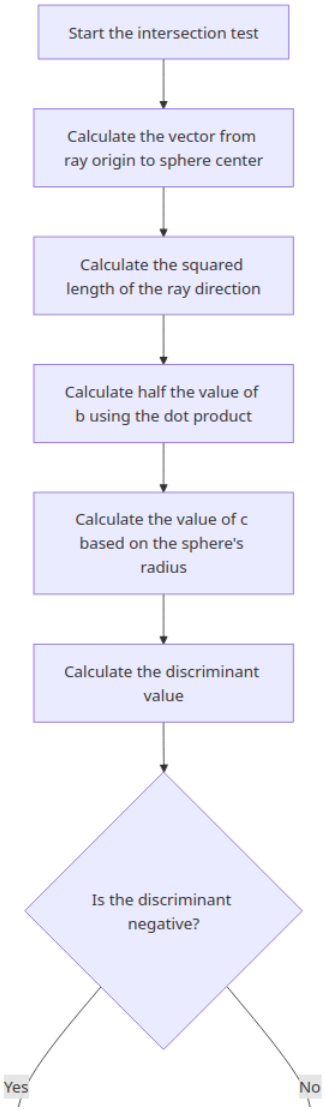
Many of these classes make use of the DVec3 class, this is something taken from the glam crate in rust, which means I do not have to create my own 3 dimensional vector class to store geometrical information and I can have more development time dedicated towards implementing features of the ray tracer.

The HittableList class defines a list of Hittable objects which can be added or removed from the scene. This will be used to keep track of what is in the scene as it is being edited by the user. A large majority of the scene files created and saved by the user will be the information stored within the scene's HittableList object.

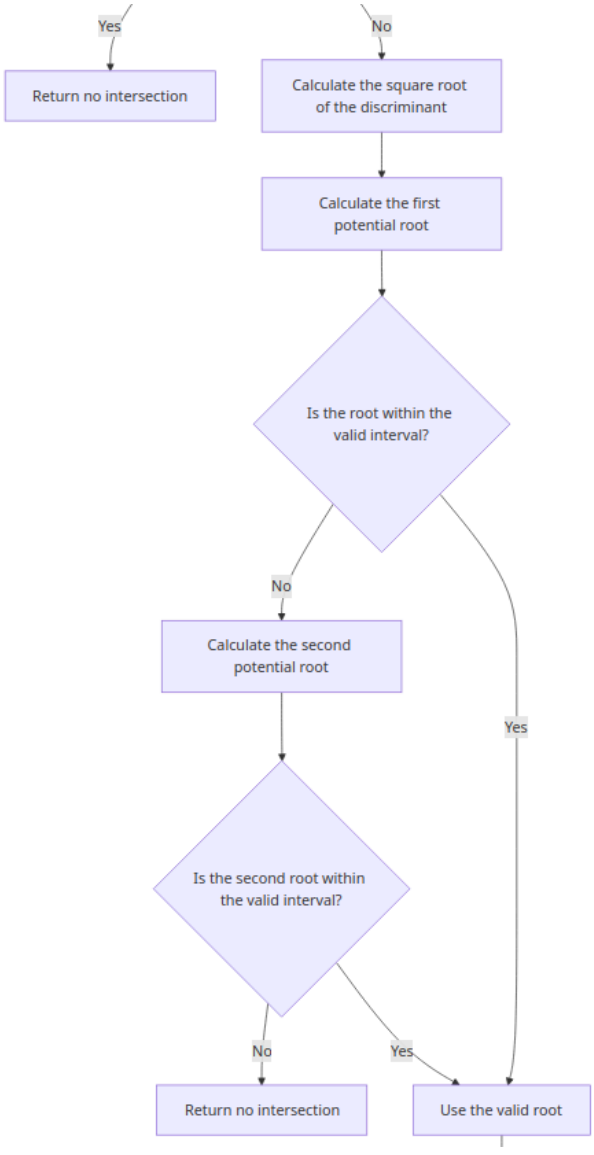
Sphere Intersection Flow Chart

One of the more mathematically complicated areas of the renderer towards the beginning of development will be the sphere intersection code. Because of this, I have decided to make a flowchart for how this function will work before I go into the development stage of my project and attempt to implement it in code.

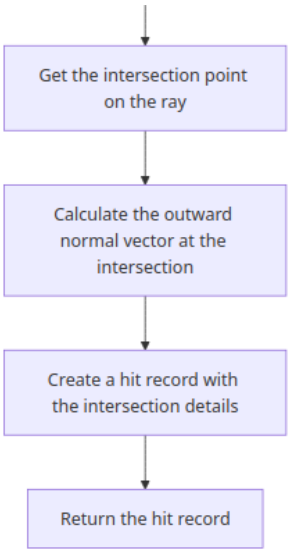
In the following flowchart, the values a, b and c represent the coefficients of the x^2, x^1 and x^0 terms of the quadratic equation explained in the research section covered earlier.



In the first section of the algorithm, we use the equation of the sphere (its centre and radius) - as well as the vector from the rays origin to the centre of the sphere - to obtain values of a, b and c for the quadratic equation we are constructing. We then calculate the discriminant of this equation and determine if it is negative or positive. Based on this, we can make judgments (if it's negative then there are no roots and therefore no intersections) and further calculations to decide whether there is an intersection or not.



In the next section of the algorithm, if the discriminant is positive we need to check if at least one of the roots is within the valid interval for there to be an intersection between the sphere and the ray. If there is then we can move on to the final stage of the algorithm.



If we can find a valid root, the next steps are to find where on the ray the intersection occurs, the outward normal vector of the point on the sphere where the intersection occurs, and create a HitRecord with this information which we can then return.

Project Plan

I will be following the agile development methodology which breaks down the project into 3 sprints, or iterations. At the end of each sprint I will perform tests to help guide me in my next sprint analysis - so I can make informed decisions on how to prioritise my time moving forward.

Sprint 1

For sprint 1, my goal will to create semi-basic working versions of each top layer section of my decomposition - however I expect that I may have to wait until sprint 2 to have a working version for the 3D environment as it relies heavily on input from the data manager - which relies on the 3D environment portion of the user interface. This will include features such as the **login**, **dashboard** & **object manager** (for the interface) and **path tracer**, **textures** & **positionable camera** (for the renderer). Finally, I will develop all of the data manager features which link to the ones I cover under the other sections.

Sprint 2

During sprint 2, I will be trying to complete the majority of the **3D environment** and the remainder of the other sections which don't have any outstanding dependencies. This will mean Adding in the **version control UI** and most of its data management counterpart, as well as the more niche renderer features like **motion blur** and **transformations**.

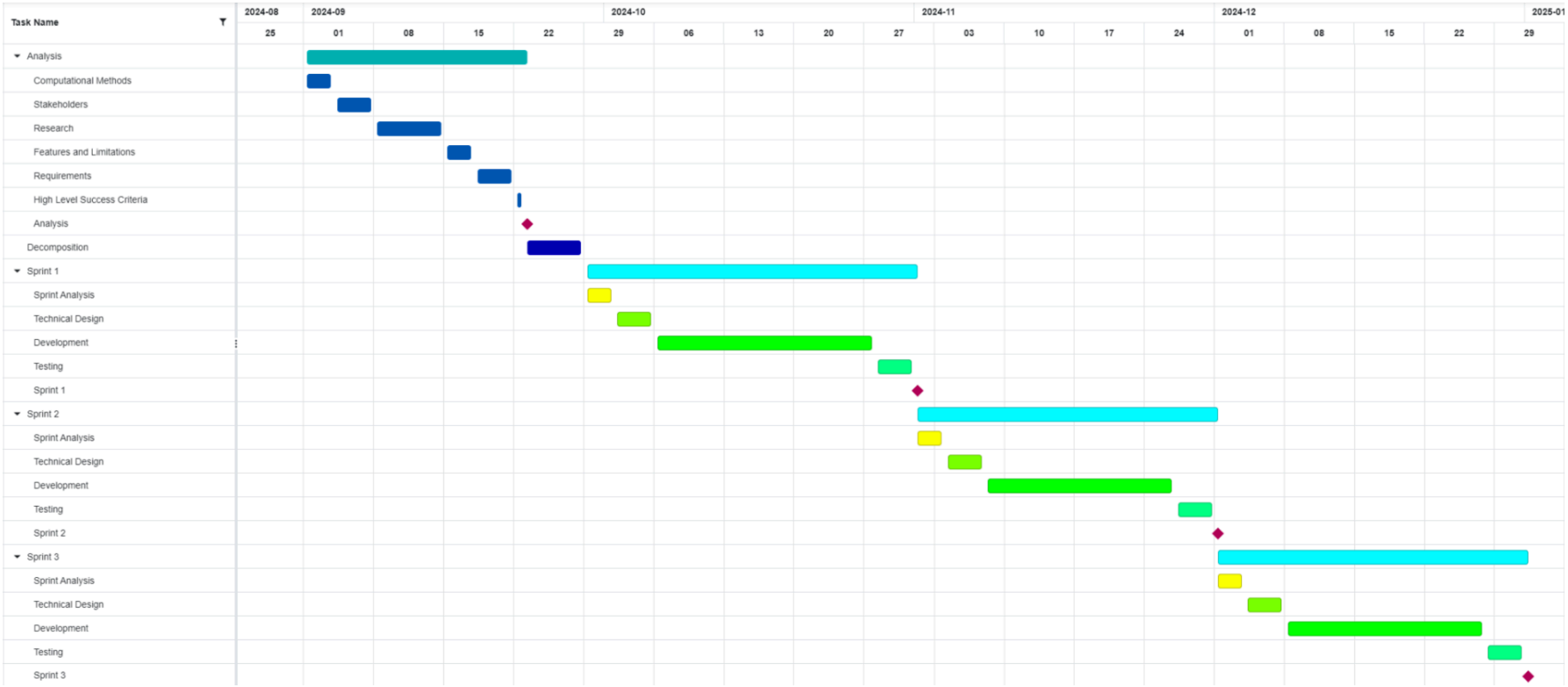
Sprint 3

Finally, for sprint 3, I plan to pull together the different independent parts of the software such as the renderer and interface into one combined program (as they will be developed and run mostly separately until this point) and update the **data manager** to work with all parts at once. Additionally, I will complete any finishing touches such as **cinematic effects** or **UI improvements** which I did not have time for in the previous sprints.

Tasks

Task Name	Start	End	Duration
Analysis	2024-09-01	2024-09-23	22 days
Computational Methods	2024-09-01	2024-09-03	3 days
Stakeholders	2024-09-04	2024-09-07	4 days
Research	2024-09-08	2024-09-14	7 days
Features and Limitations	2024-09-15	2024-09-17	3 days
Requirements	2024-09-18	2024-09-21	4 days
High Level Success Criteria	2024-09-22	2024-09-22	1 day
Analysis	2024-09-23	2024-09-23	0 days
Decomposition	2024-09-23	2024-09-28	6 days
Sprint 1	2024-09-29	2024-11-01	33 days
Sprint Analysis	2024-09-29	2024-10-01	3 days
Technical Design	2024-10-02	2024-10-05	4 days
Development	2024-10-06	2024-10-27	22 days
Testing	2024-10-28	2024-10-31	4 days
Sprint 1	2024-11-01	2024-11-01	0 days
Sprint 2	2024-11-01	2024-12-01	30 days
Sprint Analysis	2024-11-01	2024-11-03	3 days
Technical Design	2024-11-04	2024-11-07	4 days
Development	2024-11-08	2024-11-26	19 days
Testing	2024-11-27	2024-11-30	4 days
Sprint 2	2024-12-01	2024-12-01	0 days
Sprint 3	2024-12-01	2025-01-01	31 days
Sprint Analysis	2024-12-01	2024-12-03	3 days
Technical Design	2024-12-04	2024-12-07	4 days
Development	2024-12-08	2024-12-27	20 days
Testing	2024-12-28	2024-12-31	4 days
Sprint 3	2025-01-01	2025-01-01	0 days

Gantt Chart



Having already completed the analysis & decomposition section of my project, I expect to be able to complete each sprint within a month which would result in a finished product by the end of December. This means I can start the evaluation process at the beginning of January.

Development

Sprint 1

Sprint Analysis

Stakeholder Input

I got both of my stakeholders together to ask what they think is most important to the software's foundations and they provided the following responses:

- ➔ *"I think it's most important that you get a working version of the renderer complete first so you can start producing images asap, that way we have more time to test and fine tune it."* - Martin

- ➔ *"I agree with that but i'd also like some of the core interface to be added in sooner rather than later so I can get an idea for how i'm actually going to use it."* - Jason

After considering both points, I have decided to first tackle the core requirements necessary to generate my first images (to a relatively high standard but without many special effects). These include most of the path tracing requirements like normals and intersections, as well as some basic texture features. In terms of the UI, I have decided to implement a few requirements involving the scene creator (as that will be interacting with the renderer) and one requirement each for the settings and version control menus.

This will give me a broad foundation to work from for the rest of the interface rather than going in detail to one specific part early on and losing sight of how the different portions interact with each other

| Requirement | Priority | Raised By | Description | Feature |

Object Manager	Must Have	Me (Jamie)	Tab with visual list of all the objects in the scene and their variable attributes on display to edit, as well as buttons to add and remove objects from the scene	Scene Creator
Positionable Camera	Must Have	Me (Jamie)	Method for user to move the camera to different positions and orientations around a scene	Scene Creator
Sphere Intersection	Must Have	Me (Jamie)	Calculating ray intersections with spheres	Path Tracing
Surface Normals	Must Have	Me (Jamie)	Calculating vectors normal to surfaces at the point of ray intersection	Path Tracing
Antialiasing	Must Have	Me (Jamie)	Taking multiple samples of a pixel at different nearby points and calculating an average colour value	Path Tracing
Diffuse Materials	Must Have	Me (Jamie)	Scattering rays randomly off objects which don't emit light in order to create a matte texture	Textures
Reflections	Must Have	Me (Jamie)	Specular scattering off light off an object	Textures
Dielectrics	Must Have	Me (Jamie)	Splitting single rays into separate reflected and refracted rays upon intersection to create transparent surfaces	Textures
Multiple Users	Must Have	Martin	Ability to sign in with different users who are working on different scenes	Version Control
Object Addition Menu	Must Have	Me (Jamie)	Pop up window when adding objects to chose their attributes, the quantity of the object you are adding and the position(s) of the object(s)	Scene Creator
Backgrounds	Must Have	Martin	Having a range of different types of backgrounds which can be applied to a scene for different effects and use cases	Scene Creator
Imports	Must Have	Martin	An option to import additional textures and backgrounds from your computer file system	Scene Creator

Import Validity	Must Have	Martin, Me (Jamie)	Checks to ensure imported texture and background files are the correct format	Scene Creator
Defocus Blur	Should Have	Me (Jamie)	Scaling blurring at different depths to recreate the "depth of field" effect caused by real cameras because they gather light through a large hole rather than a point	Effects
Texture Mapping	Should Have	Martin	Applying material effects to objects in the scene	Textures
Bounding Volume Hierarchies	Could Have	Me (Jamie)	Discarding subsets of ray calculations depending on intersections with larger bounding volumes	Path Tracing
Motion Blur	Could Have	Me (Jamie)	Creating the effect of movement in a still frame by adding blur to an object in one direction	Effects

Detailed Success Criteria

1. Object manager sidebar
    a. Sidebar collapses with the click of a button
    b. Object list updates in real time as objects are added/edited
    c. Objects can be collapsed/expanded by their attributes
    d. Visual object preview next to each item updates as attributes are changed
    e. Selecting object opens menu to edit attributes

2. Positionable Camera
    a. Camera properties tab in object manager sidebar
    b. Fields to enter camera properties (x, y, z, fov, focus points, depth of field)
    c. Scene rendered from updated view point once properties are changed

3. Sphere intersections
    a. Ray-sphere intersections roots calculated
    b. Ray colour updated to match sphere

4. Surface Normals
    a. Calculate surface normal vectors
    b. Shade object based on normal values

5. Antialiasing
    a. Generate pixels using multiple samples

6. Diffuse Materials
    a. Randomize vector angle off outward surface
    b. Alter number of rays absorbed based on material darkness property

7. Reflections
    a. Ray reflected at same angle to normal as it was incident on the surface
    b. Blur of reflection based on material metal polish property

8. Dielectrics
    a. Rays incident on a dielectric are split into a reflected and refracted ray
    b. The angle of refracted rays depend on the refractive index property of the material
    c. Rays incident with an angle larger than the critical angle are totally internally reflected

9. Multiple Users
    a. Page dedicated to registration/login required to be passed through before entering the application
    b. Fields for inputting the username and password
    c. Register button adds a user account
        i. Account details are added to user table

ii. Path to new user folder (and the subsequent sub folders/files) added to system

10. Backgrounds
    a. Options for various solid colour backgrounds
    b. Gradient background function

11. Imports
    a. Asset table updated as imports are added or edited
    b. Import file formats validated before adding to file system or asset table
    c. Added imports show up as options when editing e.g. object textures
    d. Option to select or add category to place texture under

12. Defocus Blur
    a. Changing fov widens the range of depths in focus
    b. Shifting depth of field value moves this range back and forth

13. Texture Mapping
    a. Default checker texture available
    b. Applying image texture to sphere wraps it around surface

14. Object Addition Menu
    a. Object addition button opens menu to select object attributes
    b. Object added to scene once saved
    c. Listing added to object manager once saved

15. Bounding Volume Hierarchies
    a. Ray interception is calculated using the BVH and hittable bounding boxes
    b. Renders are finished quicker using the BVH method

16. Motion Blur
    a. Moving objects sampled to produce uni-directional blur effect
    b. Speed of objects (and therefore the magnitude of their blur) depends on the object speed attribute
    c. Direction of blur depends on the object motion direction attribute

17. Dashboard
    a. User should be able to navigate between menus by clicking on their title button

Test Plan

| ID | Description & Justification | Type | Required Input | Expected Output | Success Criteria |

1.11	Navigation for object manager - allows the user to open an expanded view of their objects while editing a scene	Normal	Click expand tab button	Sidebar beam smoothly expands to full menu	Object Manager
1.12	Closing object manager - allows the user to have more screen space dedicated to viewing their scene	Normal	Click expand tab button	Sidebar menu smoothly collapses to a small beam on the side of the window	Object Manager
1.2	Object manger scene reactions - keeps the object manager constantly up to date with what is going on in the scene	Normal	Add object to scene	Object listing added to sidebar	Object Manager
1.3	Object manager dropdown - allows user to see more details about particular objects in the scene	Normal	Click dropdown arrow next to object listing	Object attributes expanded underneath	Object Manager
1.4	Object listing scene reactions - keeps object manager listing attributes up to date with what's happening in the scene	Normal	Change colour of object in scene	Object preview changes to correct colour	Object Manager
1.5	Open object attribute menu - shows user detailed display of object information	Normal	Double click on an object in the manager tab	Menu appears in the middle of screen showing attributes and larger preview	Object Manager
1.61	Sidebar tab switch (OM) - lets user navigate sidebar	Normal	Click object tab button on sidebar	Sidebar switches from camera to OM tab	Object Manager
1.62	Sidebar tab switch (camera) - lets user navigate sidebar	Normal	Click camera tab button on sidebar	Sidebar switches from OM tab to camera tab	Object Manager, Positionable Camera

2.11	New viewing position - changes camera pos when new coords given	Normal	Edit camera viewing position (valid coordinates)	Scene renders from new position	Positionable Camera
2.12	Maintain viewing position - keeps camera pos when invalid coords given	Erroneous	Edit camera viewing position (invalid coordinates)	Error message: "Invalid viewing coordinates", camera remains still	Positionable Camera
3.1	Rendering sphere - makes sure sphere intersections are being calculated correctly	Normal	Add a sphere object to the scene, click render button	Rendered image of a sphere produced	Sphere Intersections
3.2	Updating sphere colour - Ensures ray colours are matching object they collide with	Normal	Edit colour property of sphere	Rendered image of sphere matching the new colour produced	Sphere Intersections
4	Surface normal gradient - Displays the mapping between surface normal values and points on the sphere	Normal	Add a sphere object to the scene with surface normal texture, click render button	Rendered image of sphere with smooth colour gradient	Surface Normals
5	Edges with antialiasing - ensures multiple samples are taken for each pixel	Normal	Add sphere to scene and position camera close to the edge, click render	Rendered image of sphere with smooth edge between the object and background	Antialiasing
6	Shaded sphere - shows how rays are being diffusely scattered off matte surfaces	Normal	Add sphere with matte material, click render	Rendered image of matte, shaded, sphere casting a shadow	Diffuse Materials
7	Metal sphere reflection - shows how rays are being specularly scattered off smooth surfaces	Normal	Add sphere with metal material next to a sphere with matte material, click render	Rendered image of matte sphere in the reflection of the metal sphere	Reflections
8	Hollow glass sphere - Demonstrates the successful reflection and refraction of rays incident on dielectrics	Normal	Add glass sphere in front of matte sphere to scene, click render button	Rendered image of distorted matte sphere seen through the glass sphere	Dielectrics
9.11	Login success - allows users to login to an existing account.	Normal	username: validUserName password: correctPassword	User successfully authenticated, Success message: "Login success"	Multiple Users

9.12	Login fail - invalid username - prevents users from logging into a non existing account	Erroneous	username: invalidUserName password: anything	Error message: "username not found"	Multiple Users
9.13	Login fail - incorrect password - prevents users from logging into someone else's account	Erroneous	username: validUserName password: incorrectPassword	Error message: "Password Incorrect"	Multiple Users
9.14	Login fail - empty fields - stops users logging in with empty fields	No Input	username: empty password: empty	Error message: "Please fill login fields"	Multiple Users
9.21	Signup fail - invalid pass length - increases security	Erroneous	username: validUsername password: 123	Error message: "password must be at least 6 characters"	Multiple Users
9.22	Signup fail - username taken - ensures unique usernames for all accounts	Erroneous	username: existingUsername password: validPassword	Error message: "Username taken"	Multiple Users
9.3	Login button redirection - sends user into software once credentials are validated	Normal	Click login button (with valid credentials)	User redirected to dashboard	Multiple Users
10	Background updates - Ensures different kinds of backgrounds can be applied without interference with the scene	Normal	Switch between solid colour and gradient backgrounds, click render	Rendered images of the same scene with different colour/gradient backgrounds	Backgrounds
11.1	Import fail - incorrect file format - prevents users importing unsupported files as assets	Erroneous	Select invalid file format, file.invalid, to import	Error message: "Invalid format - file extension {invalid} not supported"	Imports
11.2	Import success - Ensures imported files are being added to database	Normal	Select file to import and apply it to a scene	Image rendered correctly with new asset applied	Imports
11.3	Display assets by category - Ensures assets are being assigned the correct categories on saving	Normal	Save multiple assets with varying categories and output the asset table content	Assets should be grouped into textures and backgrounds as well as divided into categories under each of those	Imports

12.11	Changing depth of field - FOV - Checks FOV is being calculated and applied correctly	Normal	Render 2 identical scenes with different FOVs	Scene with larger FOV should have a larger range of depths in focus	Defocus Blur
12.12	Changing depth of field - shifting - Checks the right band of depths is being focused	Normal	Render 2 identical scenes with the DOF in different places	One scene should focus on foreground while the other focuses on the background	Defocus Blur
13.1	Texture application fail - invalid texture - Prevents crashing if invalid assets are applied	Erroneous	Apply a corrupted texture to an object in the scene, click render	Error message: "{texture}, applied to {object} is invalid", the scene does not render	Texture Mapping
13.2	Texture application success - Ensures valid textures are correctly applied to objects	Normal	Apply a valid texture asset to an object in the scene, click render	Scene renders with texture correctly applied to object	Texture Mapping
13.3	Texture resizing - Ensures the application of a texture to an object remains functional as the object's attributes change	Normal	Resize and object with a valid texture applied, click render	Scene still renders with texture correctly applied and object has been appropriately resized	Texture Mapping
14.1	Object addition button - allows user to quickly add objects into the scene	Normal	User clicks addition button	Object addition menu opens in centre of screen	Object Addition Menu
14.2	Saving object success - allows scene to be updated with new objects	Normal	Click save object button (with valid attributes)	Object added to scene & object manager list	Object Addition Menu, Object Manager
14.31	Saving object fail - invalid field values - prevents user adding objects with invalid attributes	Erroneous	Click save object button (with invalid attributes)	Error message: "Invalid field value: {fieldData}", object not added to scene	Object Addition Menu
14.32	Saving object fail - empty fields - prevents user from adding objects without necessary attributes	No Input	Click save object button (with empty fields)	Error message: "Please fill in {fieldName} field", object not added to scene	Object Addition Menu
15	Rendering large scenes - Ensures	Normal	Create a scene with 200	Scene renders in less time using	BVH

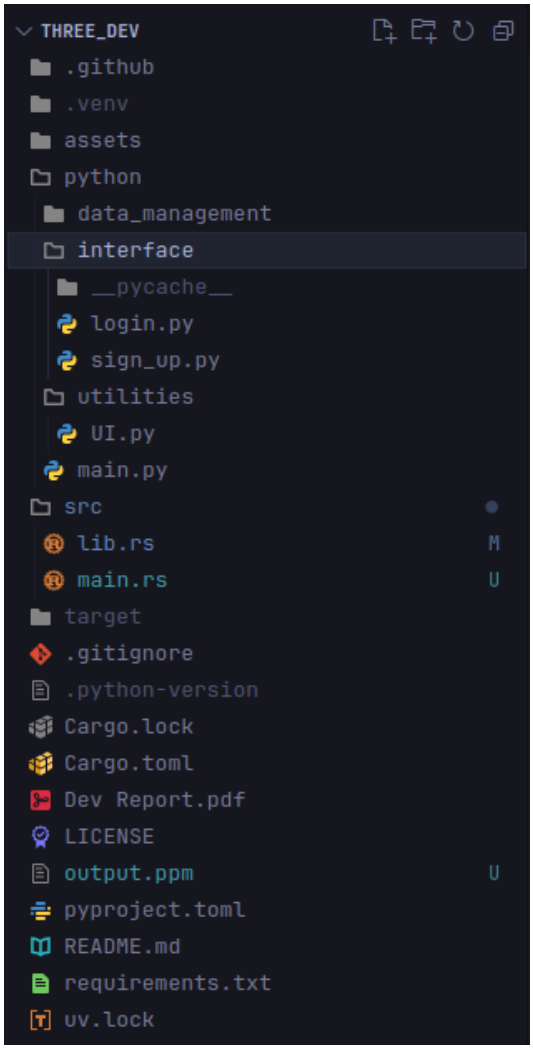
	optimisations meet usage requirements		objects spaced around, render twice (with and without BVH)	BVH than without	
16.11	Blur direction error - negative angle - ensures negative angles are not inputted	Erroneous	Negative blur direction attribute	Error message: "{angle} is not a valid blur direction - value must be > 0"	Motion Blur
16.12	Blur direction error - zero angle - ensures an angle is given	Erroneous	Blur direction set to 0	Error message: "{angle} is not a valid blur direction - value must be > 0"	Motion Blur
16.13	Blur direction error - max angle limit - ensures only principle angles are taken	Erroneous	Blur direction > 360	Error message: "{angle} is not a valid blur direction - value must be < 360"	Motion Blur
16.2	Blur direction success - 360 - ensures every angle can be displayed	Boundary	Blur direction set to 360	Image renders with object blur vertically down	Motion Blur
17.1	Navigation to gallery - allows users to access the asset gallery from the main dashboard and settings menu	Normal	user clicks on gallery button	User redirected to asset gallery	Dashboard
17.2	Navigation to settings menu - allows users to navigate to the settings menu from the main dashboard and gallery	Normal	user clicks on settings button	User redirected to settings page	Dashboard
17.3	Navigation to dashboard - allows users to access the dashboard form the settings menu and gallery	Normal	user clicks on dashboard button	User redirected to dashboard	Dashboard

NOTE: Every integer test ID corresponds to the number of its success criteria in the section above

Programming

Setup

My rust code for different aspects of the ray tracer is going to go in *src/*. This code will then be called from the python code inside *python/* when necessary. The file structure for my project is as follows

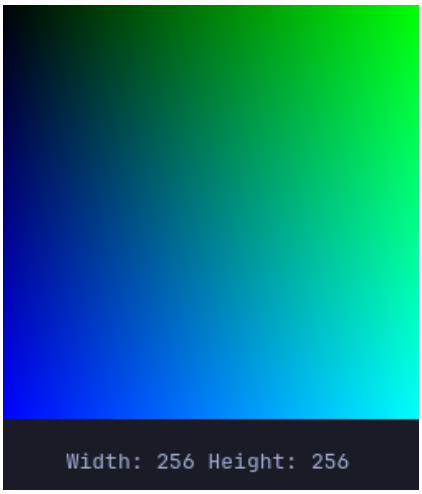


I have decided to use the uv package manager to handle the dependencies for the python aspect of my project, this means I may not need to include some files such as pyproject.toml. However, this file was introduced by maturin (open source package used to create a macro for linking python and rust - this will be explained and tested in sprint 3 when I work on linking the UI (python) to the ray tracer (rust)) so until I have an opportunity to do some testing of the rust and python working together I am going to keep both files present.

My first step for developing the ray tracer was to create an image using the PPM format. PPM is a text format in which the first 3 lines indicate the range of values being used, the dimensions of the image and the max colour value of each pixel. For the first test I just represented the x and y coordinates of each pixel, on a 256x256 image, as g and b values (and just set the r value to zero).

```rust
use itertools::Itertools;
use std::{fs, io};

// Constants
const IMG_HEIGHT: u32 = 256;
const IMG_WIDTH: u32 = 256;
const MAX_VAL: u32 = 255;

fn main() -> io::Result<()> {
    let pixels = (0..IMG_HEIGHT)
        .cartesian_product(0..IMG_WIDTH)
        .map(|(y, x)|{
            let r = 0.0;
            let g = x as f64 / (IMG_WIDTH - 1) as f64;
            let b = y as f64 / (IMG_HEIGHT - 1) as f64;
            format!(
                "{} {} {}",
                r * 255.0,
                g * 255.0,
                b * 255.0,
            )
        })
        .join("\n");
    fs::write(
        "output.ppm",
        format!(
            "P3 {} {} {} {}",
            IMG_WIDTH,
            IMG_HEIGHT,
            MAX_VAL,
            pixels,
        )
    )
}
```

The `main` function uses the cartesian product method from itertools to collect all the x and y values in our given ranges (256, 256) - this allows me to avoid working with nested loops and indices. I then map corresponding rgb values onto each point depending on its position and write the output to a .ppm file. This produces the following result:



Sphere

My next step is to send rays into the scene. For this I will be using `DVec3` from the glam library and a class to implement rays before sending them into the scene.

```rust
struct Ray {
    origin: DVec3,
    direction: DVec3,
}

impl Ray {
    fn at(&self, t: f64) -> DVec3 {
        self.origin + t * self.direction
    }
    fn colour(&self) -> DVec3 {
        let unit_direction: DVec3 =
            self.direction.normalize();
        let a = 0.5 * (unit_direction.y + 1.0);
        return (1.0 - a) * DVec3::new(1.0, 1.0, 1.0)
            + a * DVec3::new(0.5, 0.7, 1.0);
    }
}
```

In order to implement these rays, I need to define some new constants to set up the scene with a camera and viewport as explained towards the beginning of my report.

```
const IMG_WIDTH: u32 = 400;
const ASPECT_RATIO: f64 = 16.0 / 9.0;
const IMG_HEIGHT: u32 =
    (IMG_WIDTH as f64 / ASPECT_RATIO) as u32;
const MAX_VAL: u8 = 255;
const VIEWPRT_HEIGHT: f64 = 2.0;
const VIEWPRT_WIDTH: f64 = VIEWPRT_HEIGHT
    * (IMG_WIDTH as f64 / IMG_HEIGHT as f64);
const FOCAL_LEN: f64 = 1.0;
const CAMERA_CENTER: DVec3 = DVec3::ZERO;

const VIEWPORT_U: DVec3 =
    DVec3::new(VIEWPRT_WIDTH, 0., 0.);
const VIEWPORT_V: DVec3 =
    DVec3::new(0., -VIEWPRT_HEIGHT, 0.);
```

To send rays into the scene, I need to calculate the upper left pixel as well as the delta x/y vectors between pixels. The position of the upper left pixel is found by taking the top left (origin) position of the viewport and moving vertically/horizontally by the half the vertical/horizontal pixel deltas (distance between pixels). From there I can iterate over every pixel in the image and cast a ray from the centre of the camera towards the location of the pixel. Later on, I will be able to calculate any intersections the ray has with objects on its path between the camera and the viewport - the colour of the ray can then be changed to represent the shape on the image.

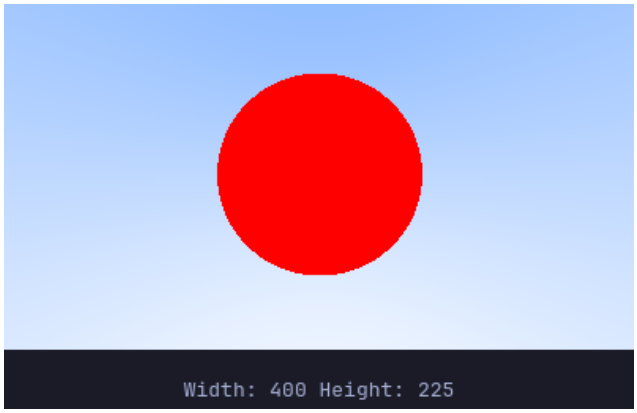
```
fn main() -> io::Result<()> {
    let pixel_delta_u: DVec3 =
        VIEWPORT_U / IMG_WIDTH as f64;
    let pixel_delta_v: DVec3 =
        VIEWPORT_V / IMG_HEIGHT as f64;

    let viewport_upper_left: DVec3 = CAMERA_CENTER
        - DVec3::new(0., 0., FOCAL_LEN)
        - VIEWPORT_U / 2.
        - VIEWPORT_V / 2.;
    let pixel00_loc: DVec3 = viewport_upper_left
        + 0.5 * (pixel_delta_u + pixel_delta_v);

    let pixels = (0..IMG_HEIGHT)
        .cartesian_product(0..IMG_WIDTH)
        .map(|(y, x)| {
            let pixel_center = pixel00_loc
                + (x as f64 * pixel_delta_u)
                + (y as f64 * pixel_delta_v);
            let ray_direction =
                pixel_center - CAMERA_CENTER;
            let ray = Ray {
                origin: CAMERA_CENTER,
                direction: ray_direction,
            };

            let pixel_colour = ray.colour() * 255.0;

            format!(
                "{} {} {}",
                pixel_colour.x, pixel_colour.y, pixel_colour.z
            )
        })
```

Now I have a ray being cast between the camera and every pixel on the viewport (image). To test this, I can add a `hit` function which uses a simple sphere to display my first ray traced image of an object. This hit function will use the discriminant of a quadratic equation formed by the combination of the sphere equation and the straight line vector equation of my ray. If the discriminant is less than 0 then we know the equation has no solutions and therefore the ray has not hit the sphere. Otherwise (if the discriminant is greater than or equal to zero, I can conclude that the ray has intersected the sphere in at least one location and the colour should therefore be changed to represent that.

This should produce a sphere on the resulting image. The reason this method works is because a quadratic can have either two, one or no solutions - this is representative of ray-sphere intersection since a ray can either miss the sphere (no solutions), hit it at a tangent/on its edge (one solution) or pass through the sphere and intersect with the outside of the front face and the inside of the back face (two solutions).

```rust
struct Ray {
    origin: DVec3,
    direction: DVec3,
}

impl Ray {
    fn at(&self, t: f64) -> DVec3 {
        self.origin + t * self.direction
    }
    fn colour(&self) -> DVec3 {
        if hit_sphere(&DVec3::new(0., 0., -1.), 0.5, self) {
            return DVec3::new(1., 0., 0.);
        };

        let unit_direction: DVec3 =
            self.direction.normalize();
        let a = 0.5 * (unit_direction.y + 1.0);
        return (1.0 - a) * DVec3::new(1.0, 1.0, 1.0)
            + a * DVec3::new(0.5, 0.7, 1.0);
    }
}

fn hit_sphere(
    center: &DVec3,
    radius: f64,
    ray: &Ray,
) -> bool {
    let oc: DVec3 = ray.origin - *center;
    let a = ray.direction.dot(ray.direction);
    let b = 2.0 * oc.dot(ray.direction);
    let c = oc.dot(oc) - radius * radius;
    let discriminant = b * b - 4. * a * c;
    discriminant >= 0.
}
```

This produces the following image:



Completed Test:

3.1	Rendering sphere - makes sure sphere intersections are being calculated correctly	Normal	Add a sphere object to the scene, click render button	Rendered image of a sphere produced	Sphere Intersections

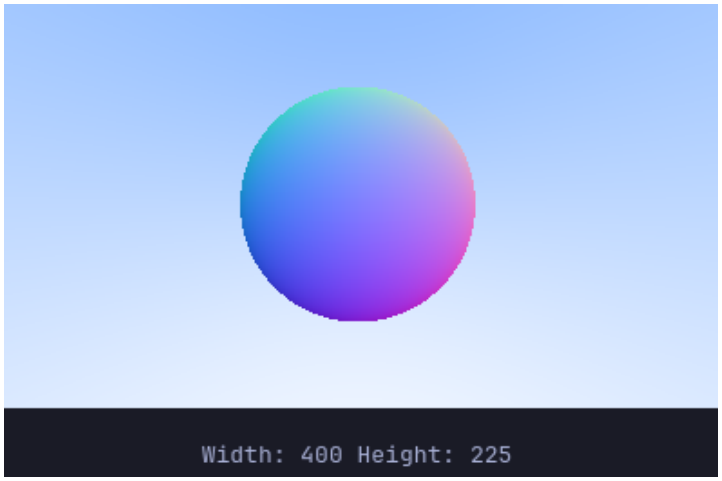
Surface Normals

Despite the object in the image being defined and calculated as a 3D sphere, it appears as a circle because I have not yet implemented surface normals which will add shading to the image. This will be my next task once I have optimised my current solution for sphere intersection as my current method could become a bottleneck when working with large numbers of spheres.

```rust
impl Ray {
    fn at(&self, t: f64) -> DVec3 {
        self.origin + t * self.direction
    }
    fn colour(&self) -> DVec3 {
        let t =
            hit_sphere(&DVec3::new(0., 0., -1.), 0.5, self);
        if t > 0.0 {
            let n_vec = (self.at(t) - DVec3::new(0., 0., -1.))
                .normalize();
            return 0.5 * (n_vec + 1.0);
        };

        let unit_direction: DVec3 =
            self.direction.normalize();
        let a = 0.5 * (unit_direction.y + 1.0);
        return (1.0 - a) * DVec3::new(1.0, 1.0, 1.0)
            + a * DVec3::new(0.5, 0.7, 1.0);
    }
}

fn hit_sphere(
    center: &DVec3,
    radius: f64,
    ray: &Ray,
) -> f64 {
    let oc: DVec3 = ray.origin - *center;
    let a = ray.direction.length_squared();
    let half_b = oc.dot(ray.direction);
    let c = oc.length_squared() - radius * radius;
    let discriminant = half_b * half_b - a * c;

    if discriminant < 0. {
        -1.0
    } else {
        (-half_b - discriminant.sqrt()) / a
    }
}
```

By refactoring the sphere intersection function and calculating normal vectors where rays intersect the sphere (excluding the edge / tangent rays), I am now able to produce a colour gradient on the sphere. Currently, the normal vector is normalised which gives it a magnitude of one, this means that each of its components will have magnitudes less than one. These components in the x, y, and z directions can be used as rgb values. Since the components have a minimum value of zero and a maximum value of 1, I can multiply each component by 255 to get the final colour.

This will later be developed to produce shading - helping the image appear 3 dimensional and more realistic.



Completed Test:

4	Surface normal gradient - Displays the mapping between surface normal values and points on the sphere	Normal	Add a sphere object to the scene with surface normal texture, click render button	Rendered image of sphere with smooth colour gradient	Surface Normals

My next steps have been to make some refactoring changes to use a less functional approach since the code is now growing and will become much more difficult to keep track of later on if I don't take a more class based approach.

I will not include all of the code that has changed as many of it is very similar but slightly re-organised. The most notable changes are the `Camera`, `HittableList` and `HitRecord` classes.

```
struct Camera {
    img_width: u32,
    img_height: u32,
    max_val: u8,
    aspect_rpoint_form_distio: f64,
    center: DVec3,
    pixel_delta_x: DVec3,
    pixel_delta_y: DVec3,
    pixel_origin: DVec3,
}
```

The `Camera` class manages all of the constants which were previously defined at the top of the file and also manages rendering the pixels to the required image format which was previously part of the ray class.

The `HitRecord` class is responsible for calculating all the information needed about a given ray-object intersection and storing that data in a useful format. This currently includes calculating and storing the surface normal information about the face that a light ray hits.

```
trait Hittable {
    fn hit(
        &self,
        ray: &Ray,
        interval: Range<f64>,
    ) -> Option<HitRecord>;
}

struct HitRecord {
    point: DVec3,
    normal: DVec3,
    t: f64,
    front_face: bool,
}
```

```rust
impl HitRecord {
    fn with_face_normal(
        point: DVec3,
        outward_normal: DVec3,
        t: f64,
        ray: &Ray,
    ) -> Self {
        let (front_face, normal) =
            HitRecord::calc_face_normal(
                ray,
                &outward_normal,
            );
        HitRecord {
            point,
            normal,
            t,
            front_face,
        }
    }
```

```rust
fn calc_face_normal(
        ray: &Ray,
        outward_normal: &DVec3,
    ) -> (bool, DVec3) {
        let front_face =
            ray.direction.dot(*outward_normal) < 0.;
        let normal = if front_face {
            *outward_normal
        } else {
            -*outward_normal
        };
        (front_face, normal)
    }
```

```rust
    fn set_face_normal(
        &mut self,
        ray: &Ray,
        outward_normal: &DVec3,
    ) {
        let (front_face, normal) =
            HitRecord::calc_face_normal(
                ray,
                outward_normal,
            );

        self.front_face = front_face;
        self.normal = normal;
    }
}
```

I can now create structs for the shapes that I intend to have in my scenes and create hittables for those objects. This has been done now for the `Sphere` struct and will later be used for quadrilaterals. The `impl` for each object will have different hit functions.

```rust
impl Hittable for Sphere {
    fn hit(
        &self,
        ray: &Ray,
        interval: Range<f64>,
    ) -> Option<HitRecord> {
        let oc = ray.origin - self.center;
        let a = ray.direction.length_squared();
        let half_b = oc.dot(ray.direction);
        let c =
            oc.length_squared() - self.radius * self.radius;

        let discriminant = half_b * half_b - a * c;
        if discriminant < 0. {
            return None;
        }
        let sqrtd = discriminant.sqrt();

        // Find the nearest root that lies in the acceptable range.
        let mut root = (-half_b - sqrtd) / a;
        if !interval.contains(&root) {
            root = (-half_b + sqrtd) / a;
            if !interval.contains(&root) {
                return None;
            }
        }

        let t = root;
        let point = ray.at(t);
        let out_normal =
            (point - self.center) / self.radius;

        let rec = HitRecord::with_face_normal(
            point,
            out_normal,
            t,
            ray,
        );

        Some(rec)
    }
}
```

I can now create the `HittableList` class which will store the hittable objects in a given region (just all the objects in the scene for now). This list can be added to or removed from as the scene is edited.

```rust
impl HittableList {
    fn clear(&mut self) {
        self.objects = vec![]
    }

    fn add<T>(&mut self, object: T)
    where
        T: Hittable + 'static,
    {
        self.objects.push(Box::new(object));
    }
}
```

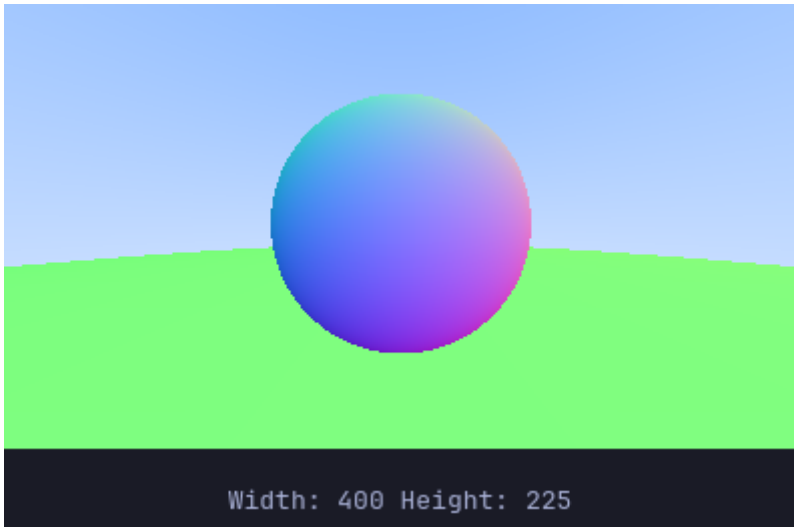
The `Hittable` impl for `HittableList` has a general hit function which will be used with objects' specific hit functions.

```rust
impl Hittable for HittableList {
    fn hit(
        &self,
        ray: &Ray,
        interval: Range<f64>,
    ) -> Option<HitRecord> {
        let (_closest, hit_record) = self
            .objects
            .iter()
            .fold((interval.end, None), |acc, item| {
                if let Some(temp_rec) = item.hit(
                    ray,
                    interval.start..acc.0,
                ) {
                    (temp_rec.t, Some(temp_rec))
                } else {
                    acc
                }
            });

        hit_record
    }
}
```

Finally, with the ability to put my sphere objects in a list, I can simplify the `main` function significantly. Later on, once the scene creator has been implemented, I will be able to use this concept of creating object structs (and managing them with a hittables list) to take input from the user about objects they would like to add to the scene and write functions to initiate these new objects in my scene rust file.

```rust
fn main() -> io::Result<()> {
    let mut scene = HittableList { objects: vec![] };

    scene.add(Sphere {
        center: DVec3::new(0., -100.5, -1.),
        radius: 100.,
    });

    scene.add(Sphere {
        center: DVec3::new(0.0, 0.0, -1.0),
        radius: 0.5,
    });

    let camera = Camera::new(400, 16.0 / 9.0);
    camera.render_to_disk(scene)?;

    Ok(())
}
```

This creates the following scene with 2 spheres of differing size, colour and position:



Completed Test:

3.2	Updating sphere colour - Ensures ray colours are matching object they collide with	Normal	Edit colour property of sphere	Rendered image of sphere matching the new colour produced	Sphere Intersections

Antialiasing

As can be seen from the last test image included, the edges of shapes are current;y not smooth at all. This is because there is a very sudden jump between the colour of the shape and the colour of the background.

My next task after the refactor is to implement anti-aliasing. This means multiple rays will be used to sample each pixel. On the boundary of an object, some nearby samples will be given the colour of the object and some will be given the colour of the background. I will then take the average of these colours and assign it to the pixel taking samples. This has the effect of creating a transition of colour on the edge of an object to the colour of the background. This is something that our eyes do automatically in real life so it increases the realism of my images.

There is a trade off which has to be made when using anti-aliasing, This is because to take multiple samples for each pixel I have to calculate more rays in total. Once I have implemented more of the later features, I can assess how well the tracer performs with different numbers of samples taken for each pixel. I will use that information to decide what the default value should be once a user creates their account.

However, different users will have different hardware availability which means they can handle different quantities of stress when it comes to sampling more rays. Because of this, the value can be changed by the user in their rendering settings in order to cater to their hardware. There will also be an option to disable anti-aliasing if the user is struggling with completing renders in a reasonable amount of time.

This function generates a random offset vector within a unit square centered at the origin of the current pixel. This vector is then used to generate a sample location within our pixel in the `get_ray` function. The use of `rng.random::<f64>()` aims for a *uniform* distribution of sample points within the unit square. While simple, this can generally produce satisfactory results. By representing a pixel bounds using a unit square, calculations become much easier while still keeping the sample space on the pixel plane. Scaling the result by the `pix_delta` vectors then creates an offset that's appropriate for the scale of this particular camera.

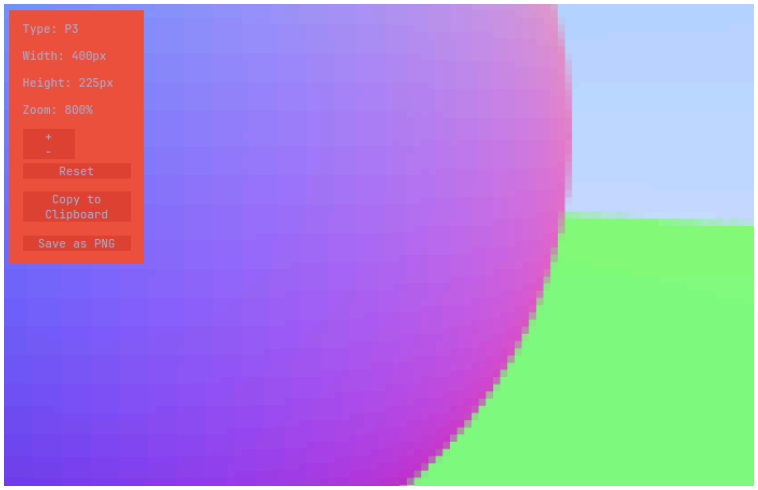
```rust
fn sample_square(&self) -> DVec3 {
        let mut rng = rand::rng();
        let px = -0.5 + rng.random::<f64>();
        let py = -0.5 + rng.random::<f64>();
        (px * self.pix_delta_x)
            + (py * self.pix_delta_y)
    }
```

This function is responsible for generating a camera ray that passes slightly within a pixel. Instead of casting a single ray through the center of the pixel, which leads to aliasing, we cast a single ray from the camera to a random location within the bounds of the pixel. By introducing random variations in the pixel sample locations we accomplish "supersampling" or "jittered sampling". By averaging the color of pixel samples using multiple rays through the pixel, we achieve a better looking antialiased pixel colour.

We now just have to call this `get_ray` function every time we want to generate a pixel to send into the scene.

```rust
fn get_ray(&self, i: i32, j: i32) -> Ray {
        let pix_center = self.pix00_loc
            + (i as f64 * self.pix_delta_x)
            + (j as f64 * self.pix_delta_y);
        let pix_sample =
            pix_center + self.sample_square();

        let ray_origin = self.center;
        let ray_direction = pix_sample - ray_origin;

        Ray {
            origin: self.center,
            direction: ray_direction,
        }
    }
```

```rust
let pixs = (0..self.img_height)
    .cartesian_product(0..self.img_width)
    .map(|(y, x)| {
        let scale_factor =
            (self.samples_per_pix as f64).recip();

        let pix_colour = (0..self
            .samples_per_pix)
            .into_iter()
            .map(|_| {
                self.get_ray(x as i32, y as i32)
                    .colour(&scene)
                    * 255.0
                    * scale_factor
            })
            .sum::<DVec3>();
```



Completed Test:

5	Edges with antialiasing - ensures multiple samples are taken for each pixel	Normal	Add sphere to scene and position camera close to the edge, click render	Rendered image of sphere with smooth edge between the object and background	Antialiasing

As can be seen above, the sphere edge is now smoother, which (when not zoomed in extremely close and/or rendered using a higher resolution) makes the image look much more realistic.

Diffuse Materials

We now need to generate a random 3D unit vector that lies on the hemisphere oriented around a given surface normal, a crucial step as diffuse reflection only happens on the hemisphere in front of the surface. It is used for calculating random directions in which to bounce the ray. The `loop` and `if vec.length_squared() < 1.` statement implements rejection sampling which repeatedly generates random vectors until one falls within the unit sphere. This simplifies the generation logic, and for small workloads, does not lead to significant waste. We use the `.length_squared` function to check vectors fall inside the unit sphere because any numbers between 0 and 1 will square to a number <= 1. This means we can check the magnitude of vectors without using the computationally expensive `sqrt()` function. This shortens render times, increasing the speed of users' workflow.

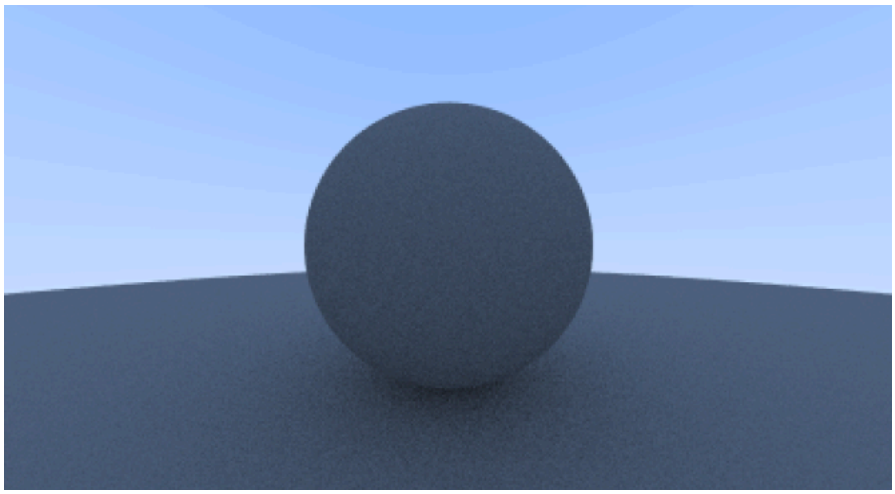
```
fn random_unit_vector() -> DVec3 {
    let mut rng = rand::rng();
    loop {
        let vec = DVec3::new(
            rng.random_range(-1.0..1.),
            rng.random_range(-1.0..1.),
            rng.random_range(-1.0..1.),
        );

        if vec.length_squared() < 1. {
            break vec.normalize();
        }
    }
}
```

Diffuse reflection is based on scattering light evenly in all directions within the hemisphere oriented by the surface's normal. This function ensures that any reflected ray (sample of the reflected light) will come from the hemisphere defined by the normal. Using random unit vectors without the check against the normal would cause rays to bounce into the surface (since they could be behind the surface), and thus produce physically incorrect visuals.

Only reflecting from the hemisphere in front of the surface contributes to the diffuse reflection. Dot product and negation are relatively cheap operations, making this function computationally efficient, and is preferred for such operations that need to be computed many times.

```
fn random_on_hemisphere(normal: &DVec3) -> DVec3 {
    let on_unit_sphere =
random_unit_vector().normalize();
    if on_unit_sphere.dot(*normal) > 0.0
    {
        on_unit_sphere
    } else {
        -on_unit_sphere
    }
}
```

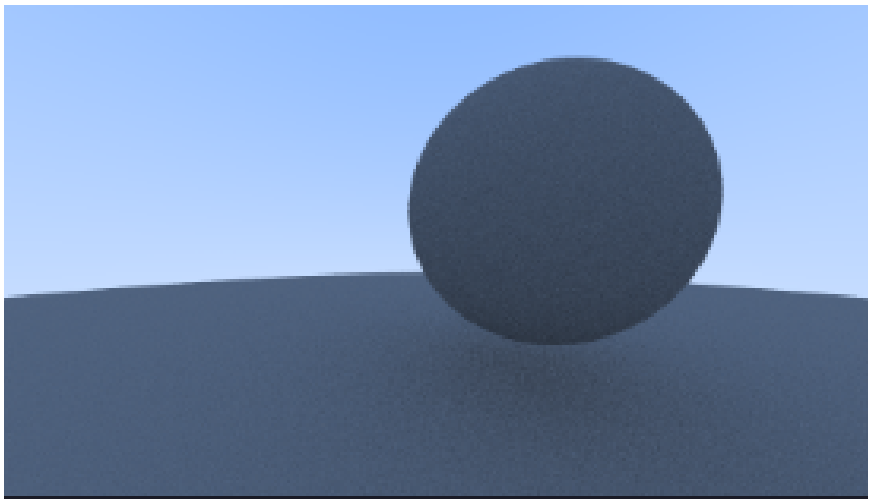
We can run this new code to render the same scene as earlier, but now looking much more 3D and realistic, with shading implemented.



Completed Test:

6	Shaded sphere - shows how rays are being diffusely scattered off matte surfaces	Normal	Add sphere with matte material, click render	Rendered image of matte, shaded, sphere casting a shadow	Diffuse Materials

We can see how the shading created by the smaller sphere onto the ground changes as I reposition it.



The next thing we can do, to improve the quality of our images, is implementing lambertian reflection. Lambert's law says *"the reflected energy from a small surface area in a particular direction is proportional to the cosine of the angle between that direction and the surface normal"*, which describes an ideal diffusely reflecting surface.

Since we will soon be implementing metals and other materials. I will create a material trait to abstract the light-surface interactions, before implementing the lambertian reflection (or metal).

```rust
trait Material {
    fn scatter(&self, ray_in: &Ray, rec: &HitRecord) -> Option<(DVec3, Ray)>;
}

struct Lambertian {
    albedo: DVec3,
}

impl Material for Lambertian {
    fn scatter(&self, _ray_in: &Ray, rec: &HitRecord) -> Option<(DVec3, Ray)> {
        let mut scatter_direction = rec.normal + random_unit_vector();

        if scatter_direction.abs().length_squared() < 1e-8 {
            scatter_direction = rec.normal;
        }

        let scattered = Ray {
            origin: rec.point,
            direction: scatter_direction,
        };

        Some((self.albedo, scattered))
    }
}
```

The `albedo` of the lambertian material describes how much of each colour component is reflected by the material.

We then add a material property to the `HitRecord` struct `Arc` is used for thread-safe reference counting - this is good practice for materials if I decide to extend to multi-threading later on in development (to improve performance). `dyn Material` is a trait object, which means it can hold any type which implements the `Material` trait (`Lambertian` or `Metal` etc). `Arc` is used to share ownership of the material between different `HitRecord` instances (without having to copy the data).

```rust
struct HitRecord {
    point: DVec3,
    normal: DVec3,
    t: f64,
    front_face: bool,
    material: Arc<dyn Material>,
}
```

Instead of calculating a new ray direction and colour directly, we will now call the `scatter` function of the object's material which was hit (could be `Lambertian` or `Metal` etc).

If the ray is scattered `Some` is returned, containing a tuple. We then recursively call `scattered.color(depth - 1, world)` to trace the scattered ray further, and we multiply the subsequent color by the `attenuation` (colour of the material). If `rec.material.scatter` returns `None` then we return `DVec3::ZERO`, black, which means no light gets reflected any further.

```rust
impl Ray {
    fn color<T>(&self, depth: i32, world: &T) -> DVec3
    where
        T: Hittable,
    {
        if depth <= 0 {
            return DVec3::ZERO;
        }

        if let Some(rec) = world.hit(self, 0.001..f64::INFINITY) {
            if let Some((attenuation, scattered)) = rec.material.scatter(self, &rec) {
                return attenuation * scattered.color(depth - 1, world);
            }
            return DVec3::ZERO;
        }

        let unit_direction = self.direction.normalize();
        let a = 0.5 * (unit_direction.y + 1.0);
        (1.0 - a) * DVec3::ONE + a * DVec3::new(0.5, 0.7, 1.0)
    }
}
```

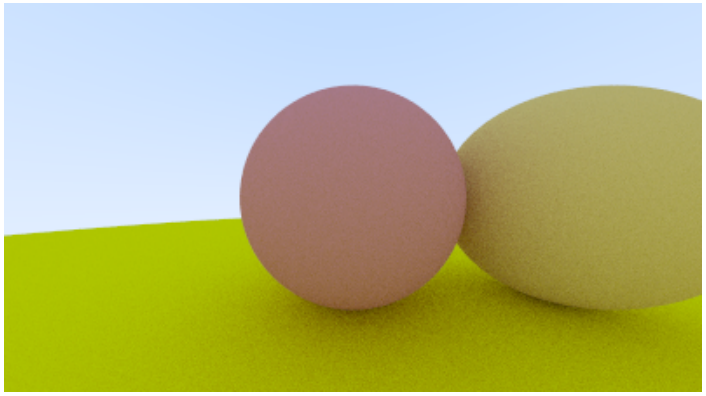
Computer displays and image formats generally use a gamma of about 2.2. This means they aren't linear in how they display color values. Without gamma correction, the linear intensity colors calculated by the ray tracer would appear too dark on a standard display. To fix this we apply a gamma of 2.0 (roughly the inverse of the display gamma) by taking the square root of each component. This makes mid-tones brighter, appearing more correct on standard displays. Using `.clamp(0.0, 0.999)` ensures the color values stay within the valid range (of 0.0 - 1.0) after gamma correction, preventing possible issues with rounding when converting into integer pixel values.

```rust
fn render_to_disk<T>(&self, world: T) -> io::Result<()>
where
    T: Hittable,
{
    let pixels = (0..self.image_height)
        .cartesian_product(0..self.image_width)
        .map(|(y, x)| {
            let scale_factor = (self.samples_per_pixel as f64).recip();
            let pixel_color = (0..self.samples_per_pixel)
                .into_iter()
                .map(|_| {
                    self.get_ray(x as i32, y as i32)
                        .color(self.max_depth as i32, &world)
                })
                .sum::<DVec3>() * scale_factor;

            let r = (pixel_color.x.sqrt()).clamp(0.0, 0.999);
            let g = (pixel_color.y.sqrt()).clamp(0.0, 0.999);
            let b = (pixel_color.z.sqrt()).clamp(0.0, 0.999);

            format!(
                "{} {} {}",
                (r * 255.999) as i32,
                (g * 255.999) as i32,
                (b * 255.999) as i32
            )
        })
        .join("\n");
}
```

We can now create materials and assign them to spheres in the main function, to produce an image of a gamma corrected, true lambertian reflected spheres.

```rust
fn main() -> io::Result<()> {
    let mut world = HittableList { objects: vec![] };

    let material_ground = Arc::new(Lambertian {
        albedo: DVec3::new(0.8, 0.8, 0.0),
    });
    let material_center = Arc::new(Lambertian {
        albedo: DVec3::new(0.7, 0.3, 0.3),
    });
    let material_right = Arc::new(Lambertian {
        albedo: DVec3::new(0.8, 0.6, 0.2),
    });

    world.add(Sphere {
        center: DVec3::new(0.0, -100.5, -1.0),
        radius: 100.0,
        material: material_ground,
    });
    world.add(Sphere {
        center: DVec3::new(0.0, 0.0, -1.0),
        radius: 0.5,
        material: material_center,
    });
    world.add(Sphere {
        center: DVec3::new(1.0, 0.0, -1.0),
        radius: 0.5,
        material: material_right,
    });

    let camera = Camera::new(400, 16.0 / 9.0);
    camera.render_to_disk(world)?;
    Ok(())
}
```

This produces the following image:



Reflections

Metal

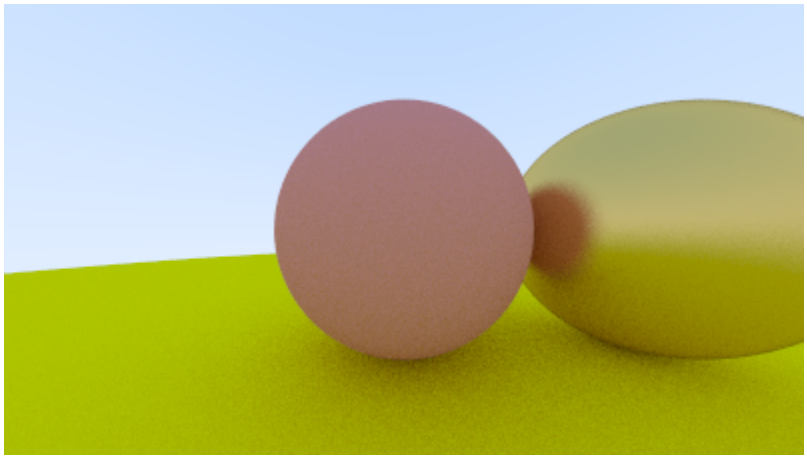
The `reflect` function calculates the direction of the reflected ray off a surface with normal `n`. This will be used in the `Metal::scatter` function to calculate the base reflection direction for metallic surfaces.

```rust
fn reflect(v: DVec3, n: DVec3) -> DVec3 {
    v - 2.0 * v.dot(n) * n
}
```

The `Metal` material struct is similar to the lambertian one, with an additional `fuzz` attribute. A `fuzz` of 0.0 represents a perfectly smooth mirror-like reflection, while a higher `fuzz` value makes the reflection more blurry or diffuse. The scattered ray direction is calculated with the perfect `reflected` direction plus a random vector inside a unit sphere scaled by `fuzz`. This randomness introduces the fuzziness, making the reflection slightly imperfect.

```rust
struct Metal {
    albedo: DVec3,
    fuzz: f64,
}

impl Metal {
    fn new(albedo: DVec3, fuzz: f64) -> Self {
        Metal {
            albedo,
            fuzz: if fuzz < 1.0 { fuzz } else { 1.0 },
        }
    }
}

impl Material for Metal {
    fn scatter(&self, ray_in: &Ray, rec: &HitRecord) -> Option<(DVec3, Ray)> {
        let reflected = reflect(ray_in.direction.normalize(), rec.normal);
        let scattered = Ray {
            origin: rec.point,
            direction: reflected + self.fuzz * random_in_unit_sphere(),
        };

        if scattered.direction.dot(rec.normal) > 0.0 {
            Some((self.albedo, scattered))
        } else {
            None
        }
    }
}
```

Using the same `main` function as earlier, but with one of the materials changed to metal, we can produce the following image, showing a metal sphere with a fuzzy reflection:



Completed Test:

7	Metal sphere reflection - shows how rays are being specularly scattered off smooth surfaces	Normal	Add sphere with metal material next to a sphere with matte material, click render	Rendered image of matte sphere in the reflection of the metal sphere	Reflections

Glass

To display glass, we need to create a dielectric material. This material will have an albedo like the other materials (to indicate how much of the rays colour is affected by the material - which can be used to display coloured tinted glass, providing more variety to the user for their scenes). Albedo is represented using a DVec3, this is because it minimizes floating point errors in ray paths - which is important for accurately maintaining the conservation of energy across numerous bounces. Additionally, the dielectric material will have a refractive index property, this controls how much the surface reflects rays and how much it refracts them. Small changes in refraction angles compound exponentially in dielectric materials so an f64 is used to maintain precision.

```rust
struct Dielectric {
    albedo: DVec3,
    refractive_index: f64,
}
```

Fresnel integrals for reflectance are very computationally expensive. Schlick's approximation is within 0.5% of the true value in most cases. `(1.0 - cosine).powi(5)` mimics reflectance curves (polynomial approximation) - this was chosen as the preferred method to balance speed of rendering times with accuracy.

The calculations for refracting rays uses the dot product of two vectors to calculate the value of the cosine of the incident angle, I have done this to avoid using the actual trigonometric functions in my code (which would have been more expensive for computational resources).

```rust
fn schlick(cosine: f64, refraction_ratio: f64) -> f64 {
    let r0 = ((1.0 - refraction_ratio) / (1.0 + refraction_ratio)).powi(2);
    r0 + (1.0 - r0) * (1.0 - cosine).powi(5)
}
```

```rust
fn refract(uv: DVec3, n: DVec3, etai_over_etat: f64) -> DVec3 {
    let cos_theta = (-uv).dot(n).min(1.0);
    let r_out_perp = etai_over_etat * (uv + cos_theta * n);
    let r_out_parallel = -(1.0 - r_out_perp.length_squared()).abs().sqrt() * n;
    r_out_perp + r_out_parallel
}
```

The `scatter` function for dielectrics is responsible for deciding whether a ray gets reflected or refracted and returning an appropriate ray with the relevant information.

The refraction ratio is adjusted (by taking its reciprocal) depending on if the ray is entering or leaving the material - this can be manipulated later on to simulate hollow or solid dielectric objects.

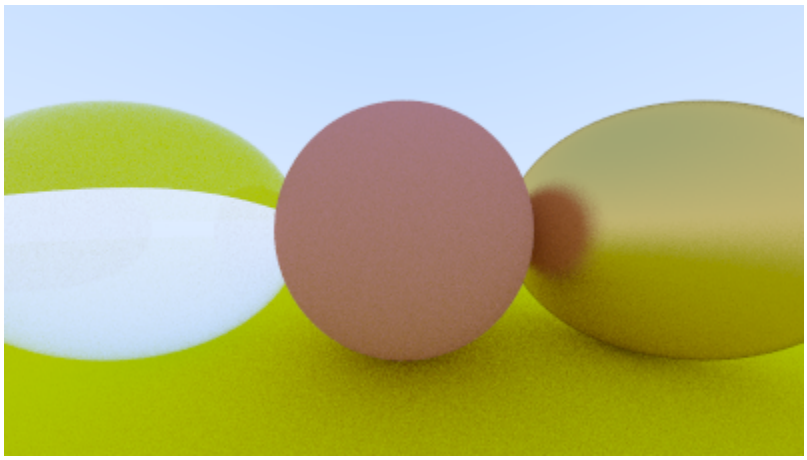
Once again, we are calculating `cos_theta` using the dot product and `sin_theta` using trigonometric angle identities in order to avoid employing the more computationally heavy built-in functions.

The next step is to use Snell's Law - comparing the angle of incidence with the critical angle - to detect if a ray undergoes total internal reflection.

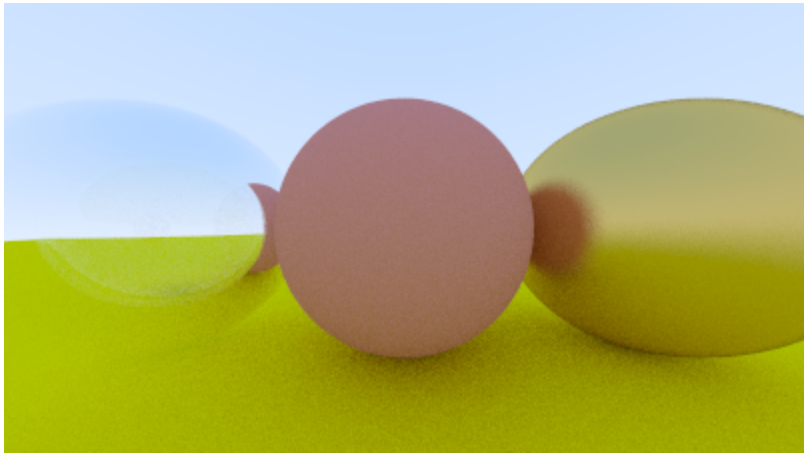
To approximate the Fresnel effect, we then call the `schlick` function discussed previously.

Finally, before returning the ray data, the reflectance is compared against a random threshold (to mimic the random behaviour of light particle interactions) and either reflected or refracted using the respective utility functions covered earlier.

```rust
impl Material for Dielectric {
    fn scatter(&self, ray_in: &Ray, rec: &HitRecord) -> Option<(DVec3, Ray)> {
        let refraction_ratio = if rec.front_face { 1.0 / self.refractive_index } else { self.refractive_index };
        let unit_dir = ray_in.direction.normalize();
        let cos_theta = (-unit_dir).dot(rec.normal).min(1.0);
        let sin_theta = (1.0 - cos_theta.powi(2)).sqrt();

        let cannot_refract = refraction_ratio * sin_theta > 1.0;
        let reflectance = schlick(cos_theta, refraction_ratio);
        let direction = if cannot_refract || reflectance > rand::rng().random() {
            reflect(unit_dir, rec.normal)
        } else {
            refract(unit_dir, rec.normal, refraction_ratio)
        };

        Some((self.albedo, Ray::new(rec.point, direction)))
    }
}
```



Rewriting the `main` function to add a glass sphere, much like we did earlier after adding the metal material, can allow us to render an image of a solid glass sphere.

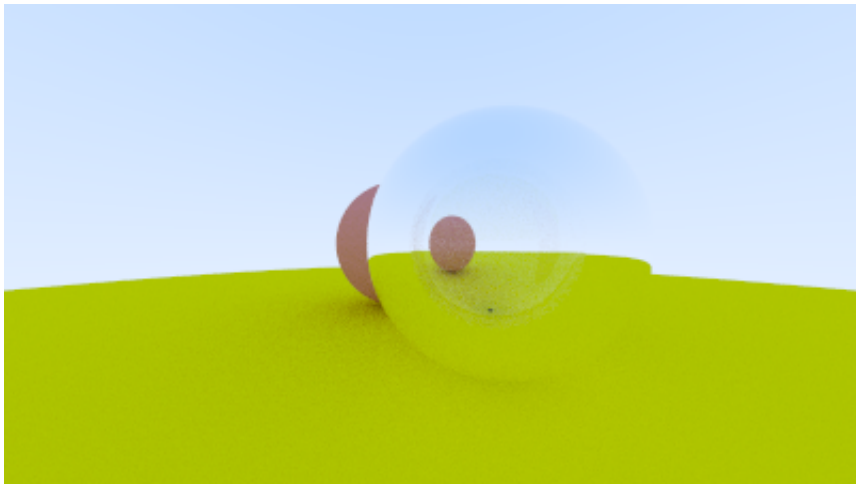


If we take the reciprocal of the refractive index of the metal sphere, then this simulates the same refraction dynamics as if the sphere were to be made of hollow glass, rather than solid glass. We will soon use this to ensure that light from objects around the sphere are being correctly calculated as they pass through the glass.

Now we can rearrange the scene to put a matte sphere directly behind the hollow glass sphere.

```
let material_ground = Arc::new(Lambertian {
        albedo: DVec3::new(0.8, 0.8, 0.0), // Matte yellow ground
    });
    let material_matte_sphere = Arc::new(Lambertian {
        albedo: DVec3::new(0.7, 0.3, 0.3), // Matte red sphere
    });
    let material_hollow_glass = Arc::new(Dielectric {
        albedo: DVec3::new(1.0, 1.0, 1.0), // Clear glass
        index_of_refraction: 1.0 / 1.5, // Inverted IOR for hollow glass effect
    });

    // Ground sphere
    world.add(Sphere {
        center: DVec3::new(0.0, -100.5, -1.0),
        radius: 100.0,
        material: material_ground,
    });

    // Solid matte sphere behind
    world.add(Sphere {
        center: DVec3::new(-0.25, 0.0, -2.0), // Behind and slightly left
        radius: 0.5,
        material: material_matte_sphere,
    });

    // Hollow glass sphere in front
    world.add(Sphere {
        center: DVec3::new(0.25, 0.0, -1.0), // In front and slightly right
        radius: 0.5,
        material: material_hollow_glass,
    });
```

This should distort the sphere, as outlined in my test plan. The spheres are slightly offset from each other in the x-direction so that the matte sphere can be seen placed behind, and reflecting through, the hollow glass one.



Completed Test:

8	Hollow glass sphere - Demonstrates the successful reflection and refraction of rays incident on dielectrics	Normal	Add glass sphere in front of matte sphere to scene, click render button	Rendered image of distorted matte sphere seen through the glass sphere	Dielectrics

Camera Properties

In order to more accurately simulate a physical camera, I will next be implementing camera positioning - allowing the user to try different viewing angles around their scene by moving the camera, rather than reconstructing the whole scene around it - and defocus blur.

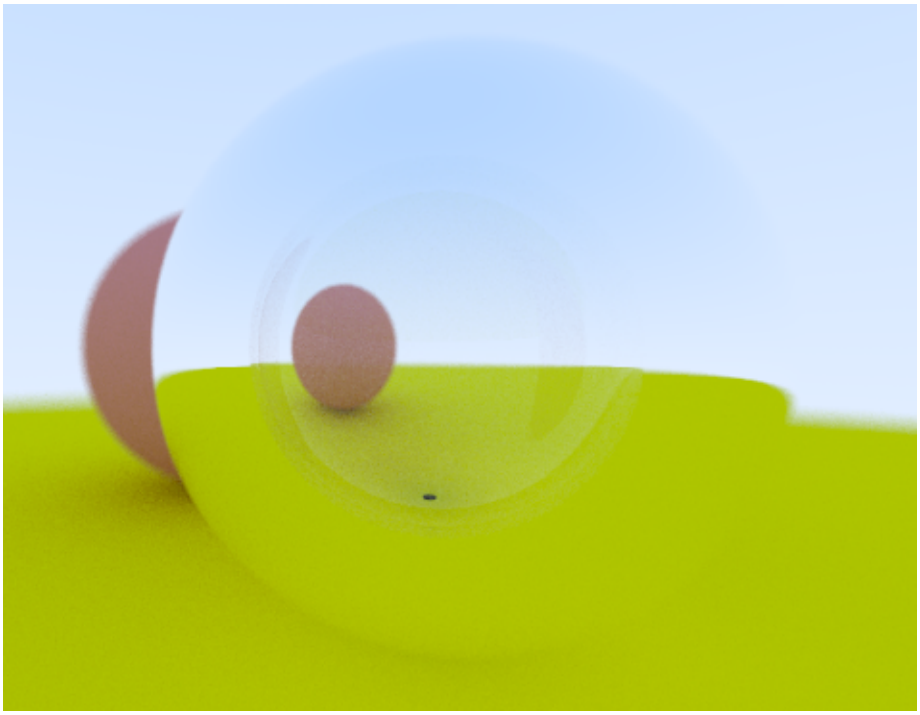
Defocus blur makes objects at different depths more or less blurry. This is based on the fact that different aperture sizes in camera lenses lead to different depths of field. Light coming from different depths hit the lens of the camera at different angles, causing the rays to converge either just in front of or just behind the focus. The human eye also experiences this which is why we have the ability to change the shape of our eye lens - giving us the ability to focus on objects at different ranges.

Some of the camera initialisation code (viewport setup) has been discussed previously. The new additions to implement the changes above are camera basis vectors and depth of field calculations.

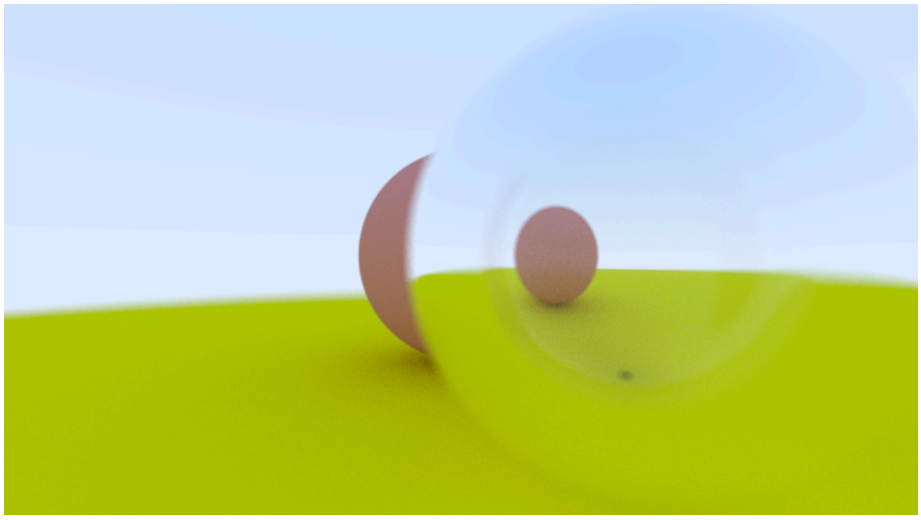
Basis vectors set up a coordinate system based on the camera's position relative to the object it is targeted towards. We achieve this by first creating a vector from the camera's 3D point position and the new `look_at` input vector. After we initialise the backwards axis, we can then utilise the cross product with the `up` vector to calculate the remaining axis. Normalisation and the cross product ensure that we end up with unit and orthogonal basis vectors.

```rust
impl Camera {
    fn new(
        image_width: u32,
        aspect_ratio: f64,
        position: DVec3,
        look_at: DVec3,
        up: DVec3,
        focus_distance: f64,
        aperture: f64,
    ) -> Self {
        let image_height = (image_width as f64 / aspect_ratio) as u32;
        let defocus_radius = aperture / 2.0;

        let w = (position - look_at).normalize();
        let u = up.cross(w).normalize();
        let v = w.cross(u);

        let viewport_height = 2.0;
        let viewport_width = viewport_height * (image_width as f64 / image_height as f64);

        let viewport_u = viewport_width * u;
        let viewport_v = viewport_height * -v;
        let pixel_delta_u = viewport_u / image_width as f64;
        let pixel_delta_v = viewport_v / image_height as f64;

        let viewport_upper_left = position - focus_distance * w - viewport_u/2.0 - viewport_v/2.0;
        let pixel00_loc = viewport_upper_left + 0.5 * (pixel_delta_u + pixel_delta_v);
```

We can run the same scene as earlier, to show how the sphere in the foreground is kept fully in focus while the sphere in the background is now slightly blurry.



If we change the values for `aperture` and `focus_distance` then we can create an image with the matte sphere focus and the glass sphere significantly blurred.



Some trial and error was required to get the correct `aperture` and `focus_distance` values for this scene. It may be useful to implement some form of sider function for these values so that the user can gradually adjust them. However, since I am not implementing real time ray tracing, I don't think it would make that much of a difference to the ease of use for the user. This is an idea that I will explore further during the evaluation stage of my project.

Completed Test:

12.11	Changing depth of field - FOV - Checks FOV is being calculated and applied correctly	Normal	Render 2 identical scenes with different FOVs	Scene with larger FOV should have a larger range of depths in focus	Defocus Blur
12.12	Changing depth of field - shifting - Checks the right band of depths is being focused	Normal	Render 2 identical scenes with the DOF in different places	One scene should focus on foreground while the other focuses on the background	Defocus Blur

The camera positioning tests will be covered in the next code section.

Personal Evaluation

I decided, during the development of the code covered so far, that it makes more sense for me to push the development of the UI to the next sprint. This is because the ray tracing is the primary focus of this project and it makes more sense to get it all completed in one go rather than over multiple sprints. Additionally, it will be much more useful to have all of the ray tracer completed before I work on the UI as I will need to use the ray tracer to test whether or not various aspects of it are functioning properly. Because of this decision, I am going to remove the UI features from this sprint and complete most of them in sprint 2, along with much of the data management. Then, in sprint 3, I will work on the scene creator and refactor my code - acting as a sprint to combine the work of the previous 2 into one application.

There is a possibility that the visual preview section of the scene creator will not have time to be implemented but that will be discussed in more detail during the evaluation stage of this project.

The plan for the rest of this sprint is to redo the sign-in page layout and aesthetics before developing the main bulk of the ray tracer.

Because most of the rendering features I develop will be dependent on the previous one, I carried out and provided evidence for these tests throughout the development and write-up process. This differs from the UI development which is much more practical to test as a final product (apart from white box tests to catch errors - and small run tests to review aesthetics - like the ones discussed earlier in this sprint) because most of the tests rely on completing a whole sign-in input process as opposed to just testing one aspect of it in a vacuum.

These are the updated requirements for sprint 1 (some may not be fully completed depending on time):

Requirement	Priority	Raised By	Description	Feature	
Sphere Intersection	Must Have	Me (Jamie)	Calculating ray intersections with spheres	Path Tracing	22/08/2024
Surface Normals	Must Have	Me (Jamie)	Calculating vectors normal to surfaces at the point of ray intersection	Path Tracing	22/08/2024
Antialiasing	Must Have	Me (Jamie)	Taking multiple samples of a pixel at different nearby points and calculating an average colour value	Path Tracing	22/08/2024
Diffuse Materials	Must Have	Me (Jamie)	Scattering rays randomly off objects which don't emit light in order to create a matte texture	Textures	22/08/2024
Reflections	Must Have	Me (Jamie)	Specular scattering off light off an object	Textures	23/08/2024
Dielectrics	Must Have	Me (Jamie)	Splitting single rays into separate reflected and refracted rays upon intersection to create transparent surfaces	Textures	23/08/2024
Quadrilaterals	Must Have	Me (Jamie)	Calculating ray intersections with quadrilaterals	Path Tracing	26/08/2024
Lights	Must Have	Me (Jamie)	Being able to include numerous light sources throughout the scene	Effects	26/08/2024
Water Reflections	Must Have	Me (Jamie)	Blurring and stretching the reflections off water as well as reducing the intensity of light reflected in order to better replicate the way light bounces of the surface of water	Textures	05/09/2024
Defocus Blur	Should Have	Me (Jamie)	Scaling blurring at different depths to recreate the "depth of field" effect caused by real cameras because they gather light through a large hole rather than a point	Effects	26/08/2024
Volumes	Should Have	Me (Jamie)	Creating the effect of translucent gasses such as smoke or fog within a scene	Effects	26/08/2024
Texture Mapping	Should Have	Martin	Applying material effects to objects in the scene	Textures	26/08/2024
Perlin Noise	Should Have	Me (Jamie)	Creating non-repeating textures such as marble using random noise generation	Textures	01/09/2024

Metallic Blur	Should Have	Me (Jamie)	Adding a smooth blur to the reflection effect seen on metallic objects to make it appear more natural and realistic	Textures	04/09/2024
Ambient Occlusion	Should Have	Me (Jamie)	Calculating a scalar value for each point on a surface to determine the amount of ambient light being reflected off it from the rest of the environment. This helps to smooth shadows, making 3D objects appear more realistic	Effects	05/09/2024
Cinematic Camera Effects	Could Have	Me (Jamie)	Effects such as bloom, lens flares, chromatic aberration & vignette	Effects	06/09/2024
Bounding Volume Hierarchies	Could Have	Me (Jamie)	Discarding subsets of ray calculations depending on intersections with larger bounding volumes	Path Tracing	30/08/2024
Subsurface Scattering	Could Have	Me (Jamie)	Rendering realistic translucent materials by reflecting some rays off the surface and scattering the rest after they have partially passed through the object	Textures	06/09/2024
BVH Optimization	Could Have	Me (Jamie)	Optimising the rendering of scenes with larger quantities of objects present	Path Tracing	05/09/2024
Tyndall Effect	Could Have	Me (Jamie)	Increasing the visibility of light as it passes through certain fluids	Effects	05/09/2024
Motion Blur	Could Have	Me (Jamie)	Creating the effect of movement in a still frame by adding blur to an object in one direction	Effects	01/09/2024

Quadrilaterals

The `Quad` sruct defines a quadrilateral parametrically using vectors rather than defining individual verticy points. I have chosen to take this approach for three primary reasons:

- ➔ **Memory:** Storing an origin point and two dimension vectors is much more memory efficient than storing an array of verticy points.
- ➔ **Textures:** This approach makes it far more intuitive to apply textures to quadrilateral objects later on.
- ➔ **Efficiency:** Using vectors simplifies the calculations required to detect ray-face intersections.

```
struct Quad {
    origin: DVec3,
    u_vec: DVec3,
    v_vec: DVec3,
    material: Arc<dyn Material>,
    normal: DVec3,
    d: f64,
    w: DVec3,
}
```

After the user inputs an origin point and dimension vectors, we can use vector products to find the rest of the required values.

Note that the direction vectors will most commonly be perpendicular to each other, but they do not have to be (if the user wants to render a parallelogram). This is why the dimensions are taken as vectors rather than just two floating point numbers representing the scale in each direction.

```
impl Quad {
    fn new(origin: DVec3, u_vec: DVec3, v_vec: DVec3, material: Arc<dyn Material>) -> Self {
        let normal = u_vec.cross(v_vec).normalize();
        let d = normal.dot(origin);
        let w = u_vec.cross(v_vec) / u_vec.cross(v_vec).dot(u_vec.cross(v_vec));

        Self {
            origin,
            u_vec,
            v_vec,
            material,
            normal,
            d,
            w,
        }
    }
}
```

The `hit` function for quadrilaterals first ensures that a ray is not parallel (or very close to parallel) with the quad front face. Doing this prevents the execution of wasted calculations for rays which will not intersect anywhere in the camera's field of view.

Once we deem it worthy to calculate intersection points with the face, there are a few approaches we can take. We could break the quad down into two triangles and check each of them, or we could try to cast rays and count intersections. These approaches work, but are computationally expensive and can be prone to numerical errors, particularly at the edges and corners.

Barycentric coordinates are a far more elegant approach. We're essentially describing any point in terms of how far it is along each of our defining vectors (`u_vec` and `v_vec`).

This links back to the justification for representing quads parametrically - if we later want to add texture coords or interpolate values across the surface, barycentric coordinates give a simple way to do this as the α and β values tell us exactly where we are on the surface.

```rust
impl Hittable for Quad {
    fn hit(&self, ray: &Ray, interval: Range<f64>) -> Option<HitRecord> {
        // Calculate intersection with the plane containing the quad
        let denom = self.normal.dot(ray.direction);

        // Ray is parallel to the plane
        if denom.abs() < 1e-8 {
            return None;
        }

        // Calculate the intersection distance
        let t = (self.d - self.normal.dot(ray.origin)) / denom;
        if !interval.contains(&t) {
            return None;
        }

        // Calculate the intersection point
        let intersection = ray.at(t);
        let planar_hitpt = intersection - self.origin;

        // Calculate barycentric coordinates
        let alpha = self.w.dot(planar_hitpt.cross(self.v_vec));
        let beta = self.w.dot(self.u_vec.cross(planar_hitpt));

        // Check if the point lies within the quad
        if alpha < 0.0 || alpha > 1.0 || beta < 0.0 || beta > 1.0 {
            return None;
        }

        Some(HitRecord::new(
            ray,
            intersection,
            t,
            self.normal,
            Arc::clone(&self.material),
        ))
    }
}
```

This allows us to create 2D (illustrated by the change in camera angle) quadrilaterals in our scene:

After these changes, I can also begin to represent the floor as a quad, rather than an extremely large sphere. This is not only more accurate for object positioning on the ground but is also less computationally expensive.

Completed Tests:

2.11	New viewing position - changes camera pos when new coords given	Normal	Edit camera viewing position (valid coordinates)	Scene renders from new position	Positionable Camera

The following test (which was scheduled for sprint 1) has now been moved to sprint 3. This is because the rust code for the ray tracer does not keep any record of scene properties between scene changes to fall back on. The python code around the scene creator will be responsible for this instead:

2.12	Maintain viewing position - keeps camera pos when invalid coords given	Erroneous	Edit camera viewing position (invalid coordinates)	Error message: "Invalid viewing coordinates", camera remains still	Positionable Camera

Since quadrilaterals (and some other requirements) were not originally intended to be part of sprint 1, and so are not part of the test plan table, I will describe the tests that they complete under the "Completed Test(s)" sections.

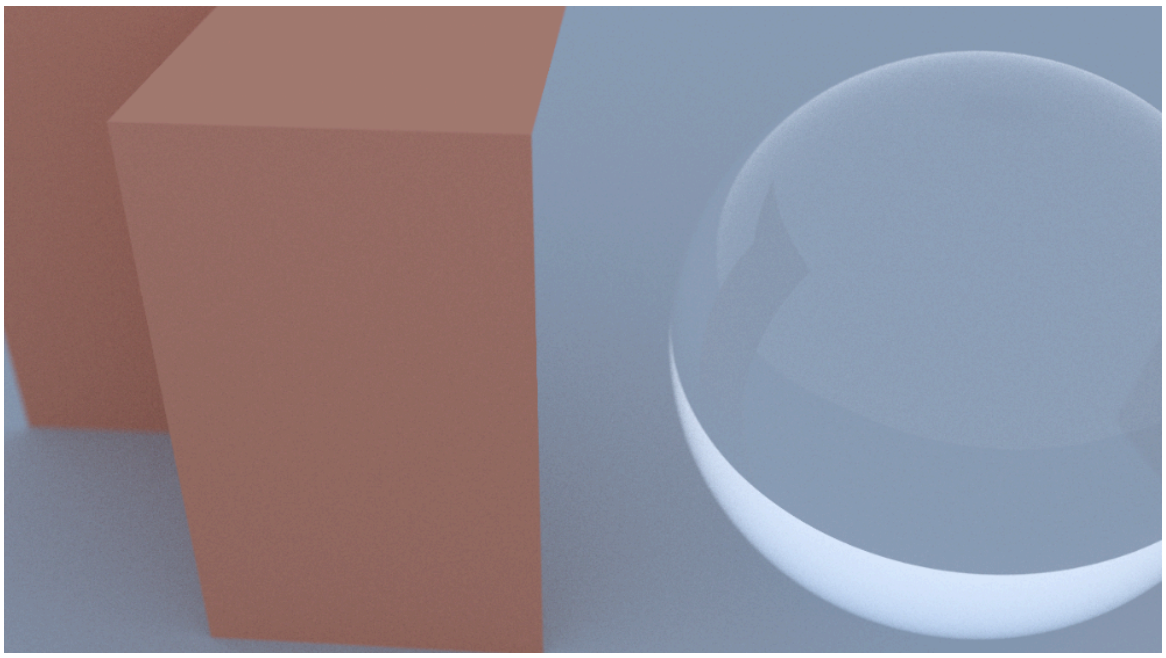
This output completes the test for displaying a 2D quadrilateral placed on the ground of the scene - related to the "Quadrilaterals" requirement in the "Path Tracing" feature. Normal data input (origin, dimension vectors and material given). The expected output - matte rectangular 2D quad displayed - has been produced.

Cuboids

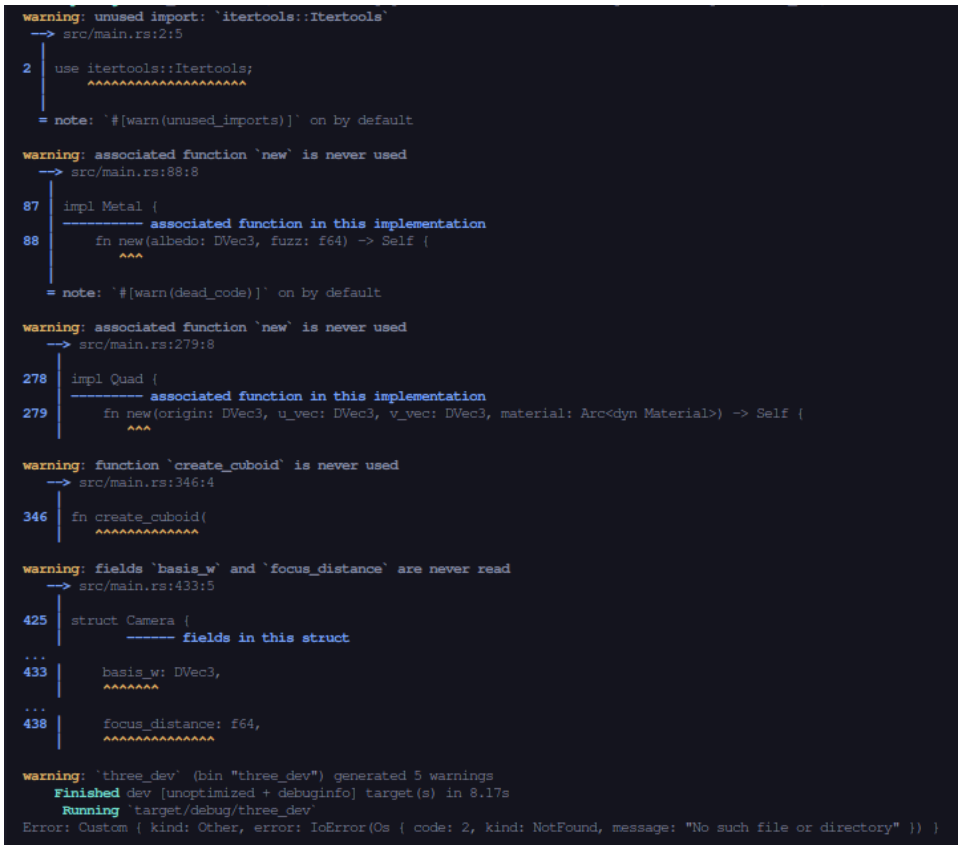
Unless the object is made of glass (in which case `refractive_index` manipulation will handle it), no ray calculations need to take place within a 3D quadrilateral. Because of this, the best way to create a cuboid is just to arrange 6 `Quad`s into a 3D structure. For this, I will create a helper function that can then be called in the main function to render a 3D cuboid shape.

This function takes similar information to what is required when placing a sphere into the scene. It then places a `Quad` into the scene for each face of the cuboid described by this inputted data.

Utilising this new helper function allows us to place cuboids into the scene



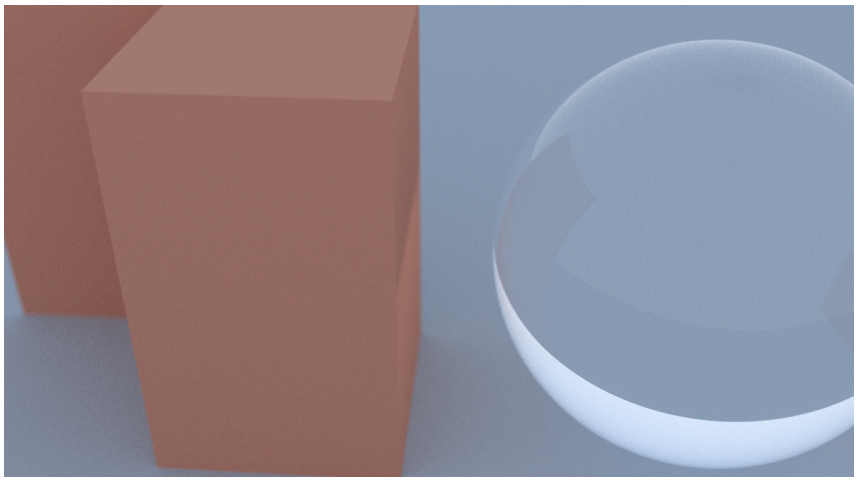
As can be seen above, the first time I tried to implement this, the right side of one of the cuboids was missing (which is a test fail). I also received the following output in the terminal:



```
warning: unused import: `itertools::Itertools`
 --> src/main.rs:2:5
  |
2 | use itertools::Itertools;
  |     ^^^^^^^^^^^^^^^^^^^^

  = note: `#[warn(unused_imports)]` on by default

warning: associated function `new` is never used
  --> src/main.rs:88:8
   |
87 | impl Metal {
   | ---------- associated function in this implementation
88 |     fn new(albedo: DVec3, fuzz: f64) -> Self {
   |        ^^^

   = note: `#[warn(dead_code)]` on by default

warning: associated function `new` is never used
   --> src/main.rs:279:8
    |
278 | impl Quad {
    | --------- associated function in this implementation
279 |     fn new(origin: DVec3, u_vec: DVec3, v_vec: DVec3, material: Arc<dyn Material>) -> Self {
    |        ^^^

warning: function `create_cuboid` is never used
   --> src/main.rs:346:4
    |
346 | fn create_cuboid(
    |    ^^^^^^^^^^^^^

warning: fields `basis_w` and `focus_distance` are never read
   --> src/main.rs:433:5
    |
425 | struct Camera {
    |        ------ fields in this struct
...
433 |     basis_w: DVec3,
    |     ^^^^^^^
...
438 |     focus_distance: f64,
    |     ^^^^^^^^^^^^^^^

warning: `three_dev` (bin "three_dev") generated 5 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 8.17s
     Running `target/debug/three_dev`
Error: Custom { kind: Other, error: IoError(Os { code: 2, kind: NotFound, message: "No such file or directory" }) }
```

In order to remedy this issue, I realised that I had an issue with the order that arguments were being passed and parameters were being received. This caused two of the inputted values to the `Camera` struct to get misplaced. Fixing this - as well as catching an incorrect negative sign on a, “Right face”, `world.add` call vector - allowed me to accurately display the two quadrilaterals in the scene and pass the test.

```rust
fn create_cuboid(
    center: DVec3,
    dimensions: DVec3,
    material: Arc<dyn Material>,
    world: &mut HittableList,
) {
    // Calculate half-dimensions for easier positioning
    let half_width = dimensions.x / 2.0;
    let half_height = dimensions.y / 2.0;
    let half_depth = dimensions.z / 2.0;

    // Front face
    world.add(Quad::new(
        center + DVec3::new(-half_width, -half_height, half_depth),
        DVec3::new(0.0, dimensions.y, 0.0),
        DVec3::new(dimensions.x, 0.0, 0.0),
        Arc::clone(&material),
    ));

    // Back face
    world.add(Quad::new(
        center + DVec3::new(-half_width, -half_height, -half_depth),
        DVec3::new(0.0, dimensions.y, 0.0),
        DVec3::new(-dimensions.x, 0.0, 0.0),
        Arc::clone(&material),
    ));

    // Right face
    world.add(Quad::new(
        center + DVec3::new(half_width, -half_height, half_depth),
        DVec3::new(0.0, dimensions.y, 0.0),
        DVec3::new(0.0, 0.0, -dimensions.z),
        Arc::clone(&material),
    ));

    // Left face
    world.add(Quad::new(
        center + DVec3::new(-half_width, -half_height, -half_depth),
        DVec3::new(0.0, dimensions.y, 0.0),
        DVec3::new(0.0, 0.0, dimensions.z),
        Arc::clone(&material),
    ));

    // Top face
    world.add(Quad::new(
        center + DVec3::new(-half_width, half_height, -half_depth),
        DVec3::new(dimensions.x, 0.0, 0.0),
        DVec3::new(0.0, 0.0, dimensions.z),
        Arc::clone(&material),
    ));

    // Bottom face
    world.add(Quad::new(
        center + DVec3::new(-half_width, -half_height, -half_depth),
        DVec3::new(dimensions.x, 0.0, 0.0),
        DVec3::new(0.0, 0.0, dimensions.z),
        material,
    ));
}
```



Completed Tests:

This output completes the 3D cuboid test. Normal data inputted (cuboid centre and dimensions given to cuboid helper function) - relating to the "Quadrilaterals" requirement. The expected output of a matte 3D cuboid displayed has been achieved.

Sprint 2

Sprint Analysis

Stakeholder Input

I asked both of my stakeholders to review the current state of my program (the UI section and the ray tracer section) after sprint 1. Because of each of their own experiences I decided to focus on collecting feedback from Martin in relation to the ray tracer and mostly collect feedback from Jason on the interface. This is what they had to say:

- ➔ *"These quality of the images are definitely at a good enough standard but i think the more features you add around affects and other kinds of shapes the more useful it will be for the game development industry in particular"* - Martin
- ➔ *"I quite like the interface so far, it's pretty easy to use but there isn't very much of it right now so i'd probably need more to judge it properly*" - Jason

Because Martin is mostly happy with the renderer and just thinks the more development the better, I have decided to wait until sprint 3 to see how much time I will have left after developing more core features of the app before I continue it any further. Jason's main concern is the lack of progress on the general UI at this point - which was to be expected after my short evaluation during sprint 1 which led to me postponing the development of UI features until this sprint.

Based on this feedback, I will continue with my current plan of using this sprint to develop the core UI - including the sign-in page and dashboard menus. My original plan for the 3 sprints mentioned data management as its own chunk of development but because of the realisations made during sprint one (which lead to the implementation of the utility methods used throughout the code) I have decided to just develop the data management features as and when they are needed for the features I am developing. Since I am doing a lot of the UI to do with project and user management in this sprint, a lot of data management will be covered as well.

| Requirement | Priority | Raised By | Description | Feature |

Multiple Users	Must Have	Martin	Ability to sign in with different users who are working on different scenes	Version Control	01/09/2024
Savable Scenes	Must Have	Me (Jamie)	Allowing the user to save all the data for a scene (as opposed to just the final render)	Version Control	05/09/2024
Resolution Dropdown	Must Have	Me (Jamie)	Dropdown to edit the resolution of the rendered image (from a range of standard options) without changing the aspect ratio	Settings Menu	04/09/2024
Aspect Ratio Dropdown	Must Have	Jason	Dropdown to select aspect ratio of rendered image, from a range of standard options (including an option for a custom aspect ratio)	Settings Menu	04/09/2024
Texture/Background Manager	Should Have	Jason	Galleries showing all of your downloaded textures/backgrounds. Glass, steel & marble will be default materials available which are generated without using a texture map	Settings Menu	01/09/2024
Performance Toggles	Should Have	Me (Jamie)	Toggles to include or disclude effects such as shadows, metallic reflections, reflections, volumes, motion blur	Settings Menu	04/09/2024
Renders Slider	Should Have	Jason	Creating sliders to edit preferences such as depth of field, when rendering so values can be adjusted more intuitively	Settings Menu	05/09/2024
Saving Options	Should Have	Jason	Allow users to choose between overwriting the current file and creating a new file with the updated content when saving projects	Version Control	09/09/2024
Version Descriptions	Should Have	Me (Jamie), Martin	Allowing the user to add a short comment or description to each version of a project as they save it - similar to Git's commit messages	Version Control	09/09/2024
Gallery Searching	Should Have	Jason	Assigning multiple search terms to each texture/background to make searching more intuitive	Settings Menu	09/09/2024
Chronological Versions	Should Have	Me (Jamie), Martin	Showing the dates on which versions of a projet were saved and displaying them in reverse chronological order (top version most recent)	Version Control	10/09/2024

Merging Scenes	Could Have	Me (Jamie)	Merging multiple different scenes from the same user or across users	Version Control	06/09/2024
Project Branching	Could Have	Me (Jamie), Martin	Allowing users to make numerous version branches from different saves along the main branch	Version Control	10/09/2024

Detailed Success Criteria

1. Multiple Users
    a. Page dedicated to registration/login required to be passed through before entering the application
    b. Fields for inputting the username and password
    c. Register button adds a user account
        i. Fields validated
        ii. Account details are added to user table
        iii. Path to new user folder (and the subsequent sub folders/files) added to system
    d. Login button redirects user to dashboard
        i. Verifies fields filled correctly
2. Dashboard
    a. User should be able to navigate between menus by clicking on their title button
3. Asset Manager
    a. Imported assets listed alphabetically
    b. Image preview above each asset title
    c. Search bar to look for assets (by title or category)
4. Settings Menu
    a. Scrollable menu to navigate preferences under different subtitles
    b. Altered settings save to database
5. Version Control Menu
    a. Scrollable menu with projects arranged in a grid of cards
    b. New project button which opens project creation menu
    c. Launch project button for each project which opens the scene creator

Test Plan

| ID | Description & Justification | Type | Required Input | Expected Output | Success Criteria |

9.11	Login success - allows users to login to an existing account.	Normal	username: validUserName password: correctPassword	User successfully authenticated, Success message: "Login success"	Multiple Users
9.12	Login fail - invalid username - prevents users from logging into a non existing account	Erroneous	username: invalidUserName password: anything	Error message: "username not found"	Multiple Users
9.13	Login fail - incorrect password - prevents users from logging into someone else's account	Erroneous	username: validUserName password: incorrectPassword	Error message: "Password Incorrect"	Multiple Users
9.14	Login fail - empty fields - stops users logging in with empty fields	No Input	username: empty password: empty	Error message: "Please fill login fields"	Multiple Users
9.21	Signup fail - invalid pass length - increases security	Erroneous	username: validUsername password: 123	Error message: "password must be at least 6 characters"	Multiple Users
9.22	Signup fail - username taken - ensures unique usernames for all accounts	Erroneous	username: existingUsername password: validPassword	Error message: "Username taken"	Multiple Users
9.3	Login button redirection - sends user into software once credentials are validated	Normal	Click login button (with valid credentials)	User redirected to dashboard	Multiple Users

11.1	Import fail - incorrect file format - prevents users importing unsupported files as assets	Erroneous	Select invalid file format, file.invalid, to import	Error message: "Invalid format - file extension {invalid} not supported"	Imports
11.2	Import success - Ensures imported files are being added to database	Normal	Select file to import and apply it to a scene	Image rendered correctly with new asset applied	Imports
11.3	Display assets by category - Ensures assets are being assigned the correct	Normal	Save multiple assets with varying categories and	Assets should be grouped into textures and backgrounds as well as	Imports

	categories on saving		output the asset table content	divided into categories under each of those	

17.1	Navigation to gallery - allows users to access the asset gallery from the main dashboard and settings menu	Normal	user clicks on gallery button	User redirected to asset gallery	Dashboard
17.2	Navigation to settings menu - allows users to navigate to the settings menu from the main dashboard and gallery	Normal	user clicks on settings button	User redirected to settings page	Dashboard
17.3	Navigation to dashboard - allows users to access the dashboard form the settings menu and gallery	Normal	user clicks on dashboard button	User redirected to dashboard	Dashboard

17.4	Scrollable Settings - allows user to view all settings easily	Normal	user drags scroll bar	Page scrolls up/down	Dashboard

Technical Design

As discussed and justified during the technical design section of the first sprint, I have already produced the designs for the UI features I plan to implement during this sprint. I will be referring back to those designs during the programming of this sprint as they are unchanged up to this point. I will not include a Technical Designs subtitle in the next sprint, following this same logic.

Programming

During this section and future sprint programming sections, when python classes are said to perform certain actions - I am usually referring to their init() methods unless otherwise specified.

Starting with just the interface aspect of it (as opposed to the data management side), I created classes to manage the key UI elements which governed smaller independent elements. These classes then call upon each other to house frames and widgets inside larger ones.

Main.py

In `main.py` (where the app is run from) I have an `Application` class which is there to create a window object for the app to run on. This window object does not have to be in a class, but functional approaches are more suited to smaller projects which don't have many different frames or windows.

I will be using the customtkinter library for interface elements and PIL for adding images into the UI. I am also currently importing the `SignIn` class from the interface directory of my project in order to display the sign-in page on the widow.

```
# Library imports
import customtkinter as ctk
from PIL import Image

# Local imports
from interface.sign_in import SignIn

# Constants
APP_TITLE = "3Dev"
```

The `Application` class initialises the basic window attributes as well as setting the theme and appearance mode (these two will only represent default values once I have created a user preferences menu for the user to set these attributes themselves). This class then applies the background for the login page before initialising the `SignIn` class and providing an empty dictionary to store sign-in data which gets passed back to main.py from the sign_in module - after the user submits their information.

The theme being applied to the application is taken from the `"...cobalt.json"` file, this is a blue theme - following on from the studies discussed during the research section of the project. The specific theme may change throughout the development of my app but I plan to stick to a general blue theme (as previously discussed).

I have opted for an object oriented approach to my application because it makes inheriting from the tkinter modules far more intuitive and easy to manage. It also means that all the different UI objects and elements are far easier to distinguish between when reading the code. Finally, it means that later on in the code I can

easily create a more general class for holding the code of frequently used elements within my application to ensure that buttons and fields etc are all consistent throughout - without needing to rewrite code.

```python
class Application(ctk.CTk):
    def __init__(
        self,
        title: str = APP_TITLE,
        size: tuple = (
            950,
            950
        )
    ) -> None:
        """
        * Initializes and manages the application window

        Parameters:
        title (str): The title of the application window.
        size (tuple): The size of the application window in pixels (width, height).
        """

        super().__init__()

        ctk.set_appearance_mode("System")
        ctk.set_default_color_theme("assets/themes/cobalt.json")

        self.title(title)
        self.geometry(f"{size[0]}x{size[1]}")
        self.minsize(
            size[0],
            size[1]
        )

        background = ctk.CTkImage(
            light_image=Image.open("assets/images/light_sign_in.png"),
            dark_image=Image.open("assets/images/dark_sign_in.png"),
            size=size
        )
        self.background = ctk.CTkLabel(
            self,
            image=background,
            text=""
        )
        self.background.pack()

        self.signInData = {}

        self.signIn = SignIn(
            self,
            self.signInData
        )
        self.signIn.place(
            relx=0.5,
            rely=0.5,
            anchor="center"
        )
```

```python
def main() -> None:
    """
    * Main function to initiate the application
    """
    app = Application()
    app.mainloop()

if __name__ == "__main__":
    main()
```

The `main` function initialises the `Application` class into an 'app' object, and runs the `.mainloop()` method - executing the code for processing the window until it gets closed. Finally, there is a condition before main() gets run to prevent this script being accidentally invoked from elsewhere.

Sign In

The `SignIn` class is responsible for managing the sign-up/login frame which the user interacts with upon opening the application. This class takes data from the user when entry boxes are filled in, then passes that data (via a dictionary) back to its parent `Application` class. The Application class then uses this data to make decisions about the page which should next be shown to the user.

```
class SignIn(ctk.CTkFrame):
    def __init__(
        self,
        master,
        signInData: dict,
        **kwargs
    ) -> None:
        """
        * Creates and displays sign in page components.

        Parameters:
        master (CTk): The parent (main app) window.
        signInData (dict): A dictionary to store user data to be passed to main.
        """

        super().__init__(master, **kwargs)

        self.signInData = signInData

        self.title = ctk.CTkLabel(
            self,
            text="3Dev",
            font=(
                "Source Code Pro",
                60
            )
        )
        self.title.pack()

        self.tabs = SignInTabs(
            self,
            self.signInData
        )
        self.tabs.pack()
```

The tabs object is created using the `SignInTabs` class. This class manages all of the fields and buttons which the user interacts with inside the sign-in page.

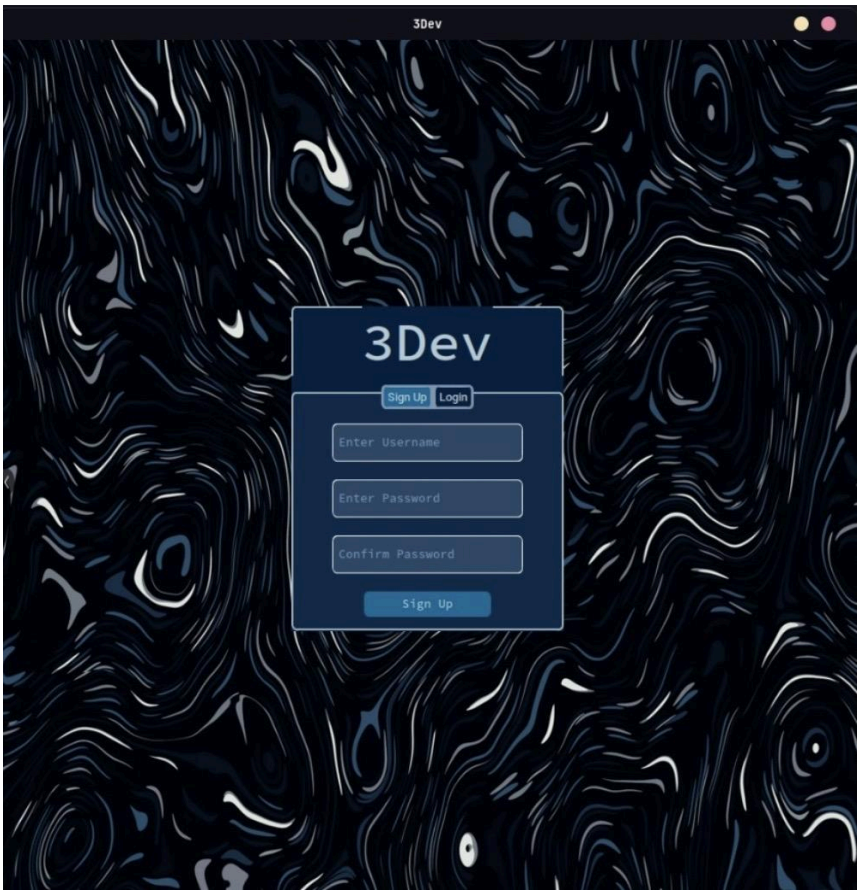
```
class SignInTabs(ctk.CTkTabview):
    def __init__(
        self,
        master,
        signInData: dict,
        **kwargs
    ) -> None:
        """
        * Constructs the "Sign Up" and "Login" tabs.

        Parameters:
        master (CTkFrame): The parent frame.
        signInData (dict): A dictionary to store user data.
        """

        super().__init__(master, **kwargs)

        self.signInData = signInData

        self.tab1 = "Sign Up"
        self.tab2 = "Login"
```

It first creates all of the entries required for the sign up page and then the submit button. Entries are created using the `createEntry()` method which I will cover later on.

```
        self.add(self.tab1)

        self.newUserEntry = self.createEntry(
            self.tab1,
            "user"
        )
        self.newUserEntry.pack(pady=10)

        self.newPassEntry = self.createEntry(
            self.tab1,
            "password"
        )
        self.newPassEntry.pack(pady=10)

        self.confPassEntry = self.createEntry(
            self.tab1,
            "passwordConf"
        )
        self.confPassEntry.pack(pady=10)

        self.signUpButton = ctk.CTkButton(
            self.tab(self.tab1),
            text=self.tab1,
            command=self.signupSubmit,
            font=(
                "Source Code Pro",
                13
            ),
        )
        self.signUpButton.pack(pady=10)
```

The same thing is then repeated for the login page and its components.

```
        self.add(self.tab2)

        self.userEntry = self.createEntry(
            self.tab2,
            "user"
            )
        self.userEntry.pack(pady=10)

        self.passEntry = self.createEntry(
            self.tab2,
            "password"
        )
        self.passEntry.pack(pady=10)

        self.loginButton = ctk.CTkButton(
            self.tab(self.tab2),
            text=self.tab2,
            command=lambda: self.loginSubmit,
            font=(
                "Source Code Pro",
                13
            ),
        )
        self.loginButton.pack(pady=10)
```



I am considering converting this section into two buttons rather than using tabs - primarily due to aesthetic reasons. This will require me to rewrite some of the `SignIn` class but the rest of the code can remain the same since I will still just be calling from the same class. I will also be changing the background image and colour theme but these will most likely become variable aspects of the UI eventually anyway, once the settings menu has been implemented.

The `createEntry()` method returns a CTkLabel in the format and styling that I am using across the whole sign-in page. This may be moved to another class later on when I start creating other pages and want to use the same styling across all of them - In which case I will make a whole dedicated class to create all of the UI components repeated across my application.

```
def createEntry(
    self,
    tab: str,
    field: str
) -> ctk.CTkEntry:
    """
    * Creates entry box to be displayed on the self.tab1 and self.tab2 tabs.

    Parameters:
    tab (bool): True if the entry is for the self.tab1 tab, False for the self.tab2 tab.
    field (str): The field for the entry box (e.g., "user", "password", "passwordConf").

    Returns:
    ctk.CTkEntry: The created entry box.
    """

    fieldDict = {
        "user": ("Enter Username", ""),
        "password": ("Enter Password", "\u2022"),
        "passwordConf": ("Confirm Password", "\u2022"),
    }

    return ctk.CTkEntry(
        self.tab(tab),
        width=210,
        height=42,
        placeholder_text=fieldDict.get(field, "")[0],
        font=("Source Code Pro", 13),
        border_color="white",
        border_width=1,
        show=fieldDict.get(field, "")[1],
    )
```

The sign up button and login button both reference their own submit methods.

The `signUpSubmit()` method validates the user inputs to ensure they meet the requirements outlined in my design section.

The first portion of this method is dedicated to carrying out all of the validation checks on the data provided by the user, and setting the values of various boolean variables to represent the outcome of these checks.

While developing this, I came across an error with the check for more than 3 consecutive ascending digits.

```
if len(pass1) in range(8, 51):
    validPassLen = True
    for i in range(len(pass1)):
        if pass1[i : i + 3].isnumeric():
            validPassDigits = not({int(pass1[j + 1]) - int(pass1[j]) for j in range(i, i+2)} == {1})
        else:
            validPassDigits = True
            # ? Attempting to identify character range issue
            print(pass1[i : i + 3])
else:
    validPassLen = False
    validPassDigits = False
```

This piece of code works by first checking the length of the password. After this it performs a nested loop to check if the difference between all the numbers in every 3-number sequence within the string is equal to 1. If this is so it means that all 3 numbers are consecutive and that the password is not valid.

The error was due to an issue with the index ranges I was using in each of the for loops. I managed to identify and fix this error using print statements to track the state of my variables after each line.

The error was fixed and the code was re-implemented into the final version of the method as shown below.

```
def signupSubmit(
        self
    ) -> None:
        """
        * Validates credentials to ensure they fit the required format - displays error messages if
        invalid, otherwise, passes credentials through to main to be handled by the data manager.
        """

        username = self.newUserEntry.get()
        pass1 = self.newPassEntry.get()
        pass2 = self.confPassEntry.get()

        if not(username and pass1 and pass2):
            self.outputMsg(
                "All fields must be filled",
                self.tab1,
                False
            )

        specChCount = lambda text: len([ch for ch in text if not ch.isalnum() and not ch.isspace()])

        userValidation = [len(username) in range(2, 51), not specChCount(username)]

        if len(pass1) in range(8, 51):
            validPassLen = True
            noDigits = True

            for i in range(len(pass1)-3):
                if pass1[i : i + 4].isnumeric():
                    noDigits = False

                    validPassDigits = not(
                        {int(pass1[j + 1]) - int(pass1[j]) for j in range(i, i+3)} == {1}
                    )
            if noDigits: validPassDigits = True
        else:
            validPassLen = False
            validPassDigits = False

        passValidation = [validPassLen, validPassDigits, specChCount(pass1) >= 2]

        validPass2 = pass2 == pass1
```

The next portion of this method handles the case for a valid input. If all credentials are valid then a suitable message - “Creating Account...” - and in the future code will be added to store these credentials into the user database.

```
if all(userValidation) and all(passValidation) and validPass2:
  self.outputMsg(
      "Creating Account...",
      self.tab1,
      True
    )

  self.passData(
      username,
      pass1,
      self.tab1
  )
```

The `outputMsg()` method is responsible for displaying a small red or green message below the submit button to give the user feedback on the issue with their credentials (if they are invalid) or what the program is doing next (if they are valid). There is currently an issue with this section of the code because if the user submits their credentials more than one time, the new output message is displayed below the previous one - rather than replacing it.

```
def outputMsg(
  self,
  msg: str,
  tab: str,
  success: bool
) -> None:
  """
  * Displays a message below the "Sign Up" or "Login" button in either red or green to indicate an error in the users
  input or what the app is doing next.

  Parameters:
  msg (str): The message to be displayed.
  tab (str): The tab where the message will be displayed.
  success (bool): A boolean indicating whether the input was valid.
  """

  signUpMessage = ctk.CTkLabel(
      self.tab(tab),
      text=msg,
      font=(
          "Source Code Pro",
          12.5
      ),
      text_color="green" if success else "red"
  )
  signUpMessage.pack()
```

Additionally, that same section of the `signUpSubmit()` method calls upon the `passData()` method. This method is responsible for updating the data in the dictionary passed through to main.py.

```python
def passData(
    self,
    username: str,
    password: str,
    tab: str
) -> None:
    """
    * Takes username and password credentials and passes them to main to be handeled by the data manager.

    Parameters:
    username (str): The username to be stored/checked.
    password (str): The password to be stored/checked.
    tab (str): The tab where the credentials were inputted (self.tab1/self.tab2).
    """

    self.signInData.update(
        {
            "username": username,
            "password": password,
            "tab": tab,
        }
    )
```

My original plan was to have main.py carry out all class operations and method calling from code in other files but this method of passing data back and forth between modules through main.py has proven to make things more difficult than it is worth. Because of this, I have decided to create a utilities directory with the class mentioned earlier (for repeated UI elements) and my data management modules. I am going to use these utility methods throughout the entirety of my code base and my main.py will still be responsible for managing and calling all of the code from my interface directory - as it is home to the master `Application` window class. This will remove the need for the `passData()` method and will also simplify the code in main.py.

The rest of the `signUpSubmit()` method is dedicated to responding to various combinations of user input errors. Each one of these has their own set of tests to ensure they function properly.

Username validation response:

```python
if not all(userValidation):
  self.newUserEntry.configure(border_color="red")

  if not userValidation[0]:
      self.outputMsg(
      "Username must be between 2 and 50 characters",
      self.tab1,
      False
  )
  else:
      self.outputMsg(
          "Username must not contain any special characters",
          self.tab1,
          False
      )
else:
  self.newUserEntry.configure(border_color="white")
```

Password validation response:

```python
if not all(passValidation):
    self.newPassEntry.configure(border_color="red")

    if not passValidation[0]:
        self.outputMsg(
            "Password must be between 8 and 50 characters",
            self.tab1,
            False
        )
    elif not passValidation[1]:
        self.outputMsg(
        "Password must not contain more than 3 consecutive numbers",
        self.tab1,
        False
    )
    elif not passValidation[2]:
        self.outputMsg(
            "Password must contain at least 2 special characters",
            self.tab1,
            False
        )
else:
    self.newPassEntry.configure(border_color="white")
```

Password confirmation validation response:

```
if not validPass2:
    self.confPassEntry.configure(border_color="red")

    self.outputMsg(
        "Passwords do not match",
        self.tab1,
        False
    )
else:
    self.confPassEntry.configure(border_color="white")
```

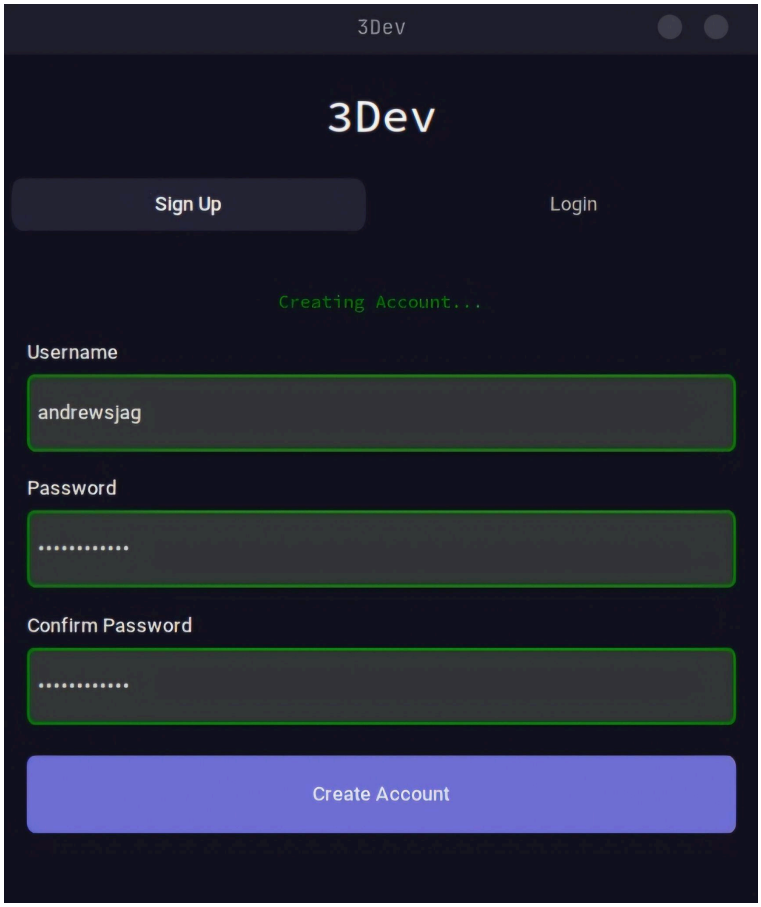
All of these display a suitable message to the user once they have submitted their credentials and also change the outline/border colour of the entry boxes to either green or red, depending on whether or not the information inputted to that field was valid.

Finally, I have the `loginSubmit()` method which is called by the login button on click. This method simply ensures that both fields have been filled before passing the data back to main.py for validation from the data manager (which will be implemented after this section of the UI).

```
def loginSubmit(
    self
) -> None:
    """
    * Outputs error message if a field is missing, otherwise, passes credentials through to main to be handled by data
    manager.
    """

    username = self.userEntry.get()
    password = self.passEntry.get()

    if not (username and password):
        self.outputMsg(
            "All fields must be filled",
            self.tab2,
            False
        )
    else:
        self.outputMsg(
            "Validating Credentials...",
            self.tab2,
            True
        )

        self.passData(
            username,
            password,
            self.tab2
        )
```

Now that I have completed the validation logic for the sign up and login pages, I am going to address the aesthetic issues mentioned earlier. I will first show the versions of this page I went through before showing the final version and how it has been implemented.

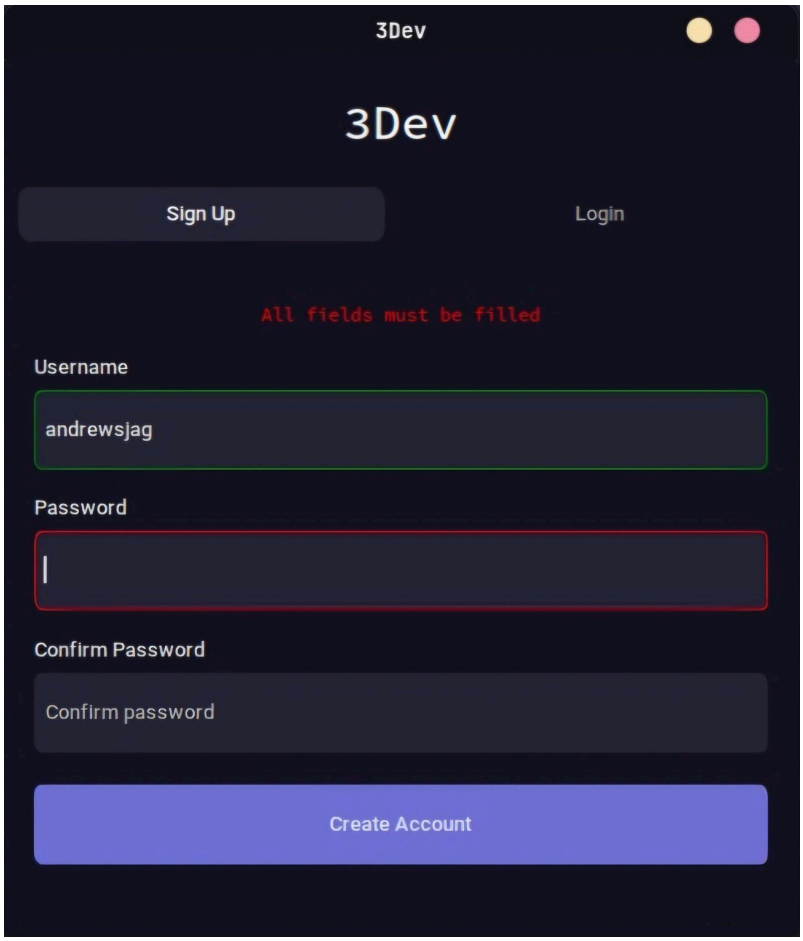
The first large change was to swap to a solid colour background and change the colour palette to a darker blue which better reflects my original designs. Additionally, as discussed earlier, I removed the tab view object and implemented two separate buttons for navigation.



In order to improve the design of this page, I enlisted the help of my stakeholder Jason, who's main role is to guide the development of the interface. He felt that there were some things making the interface look slightly off.

"I think it would be better if the entry boxes matched the rest of the screen because the gray doesn't fit very well. Also keep the green highlights with the message but I think getting rid of the normal borders will make it look cleaner"

After hearing this, I decided to change the background colour of the entry boxes to match that of the selected navigation button at the top. Additionally, I thinned the highlight borders in order to make the error indication less obtuse - Jason agreed with this decision after I showed him the new image.



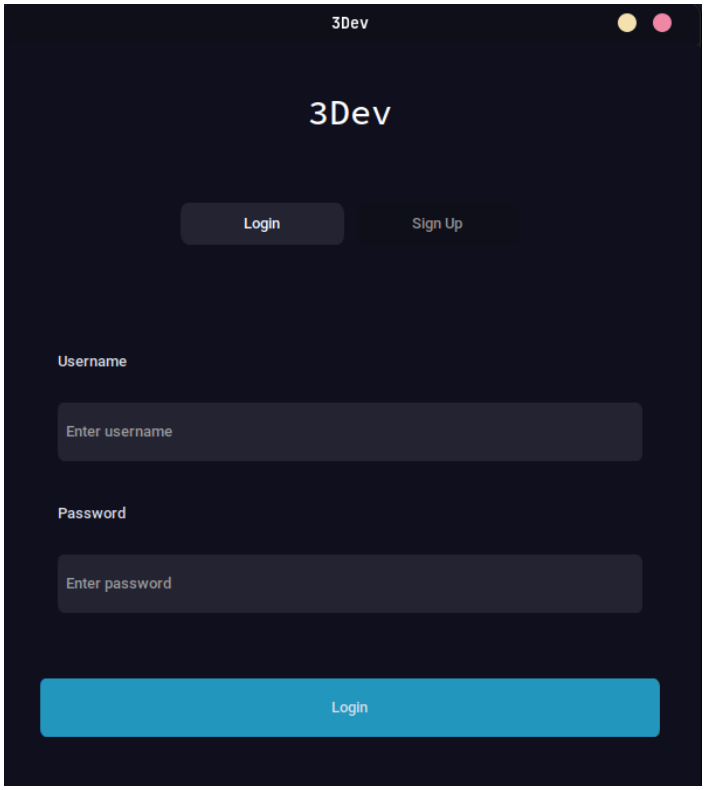
Next, he wanted to focus on the usability of the interface.

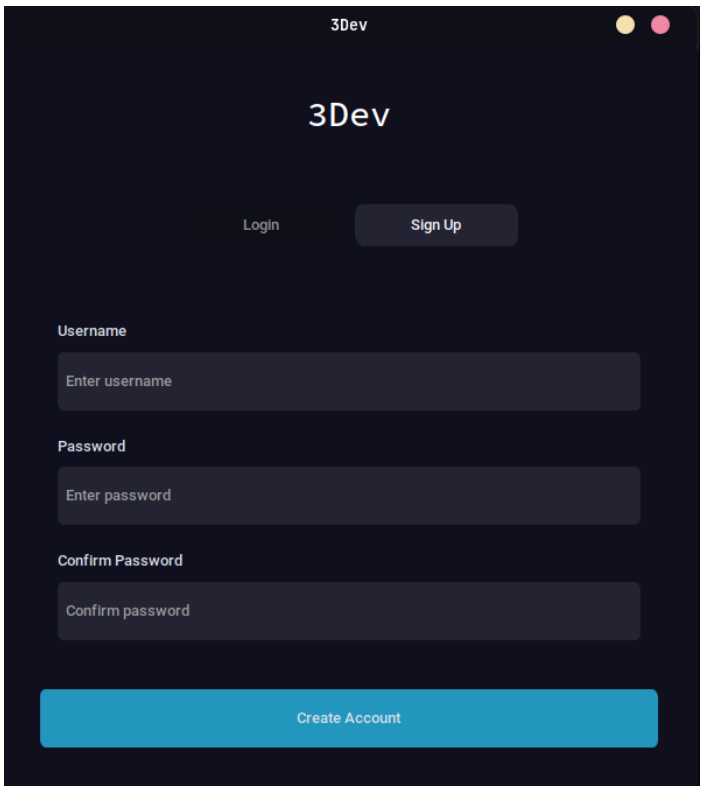
"I think there's a few issues with the layout to be honest. The login button is the same colour as the background so you can't see the clicking area for that button when it's not selected. Also I think all of the elements stretch too far to the edges of the screen, no username or password is going to be long enough to fill that space anyway with the font size you're using. The last thing is that most of the time users are going to be logging in rather than signing up so switching them around to put login on the left makes more sense to me."

There are numerous changes which need to be made after this input:

- The unselected navigation button should be more clear rather than blending into the background. My final mock-up design partially achieved this by separating the navigation buttons into their own area on the screen which made it more clear where the clicking area was - however the difficulty of implementing this with customtkinter was not worth the time it would have taken. Instead, I will change the background of the login button to be slightly darker than the main background.
- I am going to squash all of the UI elements by adding padding in the x direction, this should make the page easier to read and more aesthetically pleasing.
    - I have decided to make the navigation buttons particularly small as they contain very little information so require little space. The entries contain the most important information so will be the largest of the elements and the submit button is of key importance so I have made it larger than the navigation buttons.
- Finally, Jason has contested my previous reasoning about the sign up page being on the left as it is the first one a user interacts with. He believes that because users only have to sign up once, but use the app many times, it makes more sense to have the login page selected by default.

Below are the final versions of my sign up and login pages (I have applied the same alterations to the login section of this page).





The vertical layout of the sign-in page has been chosen based on the order in which the user will interact with its elements. The user will first read the title of the app before deciding which sign-in form they wish to fill out, they will then fill out their username and password(s) before finally clicking submit.

The sign-in page is now displayed using the `displayFrame` method. This replaces the previous lines in the `main()` function in order to create the sign in tabs.

```
def displayFrame(self, master: any, frame: ctk.CTkFrame, clear: bool):
        """
        * Displays a given frame in the application window

        Parameters:
        master (any): The parent of the given frame.
        frame (ctk.CTkFrame): The frame to be displayed.
        clear (bool): A boolean indicating whether to clear the previous frames or not.
        """
        if clear:
            for child in master.winfo_children():
                child.grid_forget()
        frame.tkraise()
```

This method is then called on the new `UserMenu` class, which contains the navigation button code, and parents the sign-up and login frames. The colour constants used in this method are imported from the UI utilities module - which also contains the `outputMsg()` method used in the `signIn` and `Login` classes.

Firstly, all of the UI elements are created and the `signIn` and `Login` classes are initialised.

Additionally, the rows and columns of the frame are configured, using customtkinter's built-in methods. This will allow me to arrange elements and frames throughout the window more accurately than with the `pack()` method, and more intuitively than with the `place()` method.

The `columnconfigure()` and `rowconfigure()` methods allow me to convert my window space into a grid, and weight columns and rows according to my positional needs. This then allows me to specify the column and row number of my UI elements when placing them onto the screen.

```
class UserMenu(ctk.CTkFrame):
    def __init__(
        self,
        master,
        **kwargs
    ) -> None:
        """
        Initialises the user menu frame and houses all of its elements

        Paramaters:
        master (Ctk): The parent window
        """
        super().__init__(master, fg_color=DARK_BLUE, **kwargs)

        self.columnconfigure((0, 1), weight=1)
        self.rowconfigure((0, 1, 2), weight=1)

        self.signup_button = ctk.CTkButton(
            self,
            text="Sign Up",
            command=lambda: self.show_menu(self.signUp),
            fg_color=DARK_BLUE,
            hover_color=HOVER_COLOUR1,
            border_width=0,
            corner_radius=8,
            text_color=UNSELECTED_COLOR,
            height=37
        )
        self.signUp = SignUp(self)

        self.login_button = ctk.CTkButton(
            self,
            text="Login",
            command=lambda: self.show_menu(self.login),
            fg_color=LIGHTER_BLUE,
            hover_color=HOVER_COLOUR1,
            border_width=0,
            corner_radius=8,
            text_color="white",
            height=37
        )
        self.login = Login(self)

        self.title_label = ctk.CTkLabel(
            self,
            text="3Dev",
            font=(
                "Source Code Pro",
                30
            ),
            text_color="white"
        )
```

Next, each of these elements will be arranged using the `grid()` method.

The parameters of the grid method describe the column and row the element should lie in, the x and y padding between the element and those around it, as well as which walls of the chosen grid square the element should "stick" to.

Passing `"e"` and `"w"` (east and west) to the login and signup buttons respectively, is what allows them to be placed close to each other in the middle of the screen - and their equal x padding (in the opposite directions) is what keeps them equally spaced about the centre line without touching each other.

The `"nesw"` argument being passed for the login and sign up frames is what causes them to fill the space in the window - even if it gets resized.

```
self.title_label.grid(row=0, column=0, columnspan=2, pady=(20, 0), sticky="ew")

self.login_button.grid(row=1, column=0, padx=(20, 5), pady=(20, 0), sticky="e")
self.signup_button.grid(row=1, column=1, padx=(5, 20), pady=(20, 0), sticky="w")

self.signUp.grid(row=2, column=0, columnspan=2, pady=30, sticky="nesw")
self.login.grid(row=2, column=0, columnspan=2, pady=30, sticky="nesw")

self.show_menu(self.login)  # Login page shown by default
```

The `show_menu()` method is called on the login page by default (it is also called inside the button command argument passed when initialising the sign-up and login navigation buttons). This method is responsible for raising the correct menu for the user to interact with, as well as altering the aesthetic state of the navigation buttons to reflect this.

```
def show_menu(self, frame):
    """
    * Raises relevant frame and alteres nav button states

    Paramaters:
    frame (CTkFrame): The frame to be raised
    """
    if frame == self.signUp:
        self.signup_button.configure(text_color="white", fg_color=LIGHTER_BLUE)
        self.login_button.configure(text_color=UNSELECTED_COLOR, fg_color=DARKER_BLUE)
    else:
        self.signup_button.configure(text_color=UNSELECTED_COLOR, fg_color=DARKER_BLUE)
        self.login_button.configure(text_color="white", fg_color=LIGHTER_BLUE)
        frame.tkraise()
```

The `SignUp` and `Login` classes contain the code for displaying the updated forms to the user. Much of this code is repeated across elements and between forms so I will show and explain one example of each of the key elements. In later sprints - if time allows it - I will refactor this code to remove as much repetition as possible, but implementation is my top priority at this time so I have allowed it on the basis that it causes no noticeable negative impact on performance.

The entries and their titles (sign-up password input in this case) are padded significantly so that they do not get too close to the edges of the screen. The entry itself is given the `"ew"` `sticky` argument so that it stretches to either side of the screen equally, while the subtitle is given the `"w"` sticky argument so that it is displayed above the left (west) hand side of the entry.

The `"\u2022"` character is passed as the `show` argument, this displays all inputted characters as dots as the user types their password, in order to prevent shouldering.

```
# Password input
self.password_label = ctk.CTkLabel(
    self,
    text="Password",
    fg_color=DARK_BLUE)
self.password_label.grid(row=4, column=0, padx=50, pady=(5, 0), sticky="w")

self.newPassEntry = ctk.CTkEntry(
    self,
    placeholder_text="Enter password",
    show="\u2022", height=50,
    fg_color=LIGHTER_BLUE,
    border_width=0)
self.newPassEntry.grid(row=5, column=0, padx=50, pady=(0, 5), sticky="ew")
```

The code for implementing the submit, “Create Account”, button is similar with some of the size, position and padding arguments altered (as well as the widget object being a `CTkButton` rather than a `CTkLabel` or `CTkEntry`).

```
# Create account button
self.create_account_button = ctk.CTkButton(
    self,
    text="Create Account",
    command=self.signupSubmit,
    height=50,
    fg_color=BUTTON_COLOUR,
    hover_color=HOVER_COLOUR2)
self.create_account_button.grid(row=10, column=0, padx=35, pady=(30, 10), sticky="ew")
```

Dashboard

The dashboard code is laid out into the `TopMenuFrame` class, which is the master of three more frame classes. These classes contain the interface for each of the sections listed in the top bar buttons - including `VersionControl`, `AssetGallery` and `Settings`.

The `DARK_BLUE` colour is passed to this frame's attributes in order to maintain design consistency with the sign-in page.

```
class TopMenuFrame(ctk.CTkFrame):
    def __init__(
        self,
        master: ctk.CTk,
        username: str,
        **kwargs
    ) -> None:
        super().__init__(master, fg_color=DARK_BLUE, **kwargs)

        # Configure grid weights
        self.grid_columnconfigure(3, weight=1)  # Make the middle space expand
        self.grid_rowconfigure(1, weight=1)

        # Initialize frames dictionary to store menu frames
        self.frames: Dict[str, ctk.CTkFrame] = {}

        # Create and store menu frames
        self.frames["version_control"] = VersionControl(self)
        self.frames["asset_gallery"] = AssetGallery(self)
        self.frames["settings"] = Settings(self)
```

Similar to the frame `fg_color` attribute, the menu buttons have been given the `BUTTON_COLOR` colour in order to match the colour scheme of the sign-in page.

In addition to the three menu buttons, the top menu bar also contains a label to display the username of the logged in user.

The username is currently just set to a `"username"` string - which will later be changed to show the actual username stored in the user table of the database.

Each button has a command argument which calls the `show_menu` method. I will soon cover the code in this method.

There are dictionaries to connect both the menu frames and their buttons to labels which get passed into the `show_menu method`.

```python
# Create buttons
self.version_control_btn = ctk.CTkButton(
    self,
    text="Version Control",
    fg_color=BUTTON_COLOUR,
    command=lambda: self.show_menu("version_control"),
    width=175,
    height=BUTTON_HEIGHT
)

self.asset_gallery_btn = ctk.CTkButton(
    self,
    text="Asset Gallery",
    fg_color="transparent",
    command=lambda: self.show_menu("asset_gallery"),
    width=175,
    height=BUTTON_HEIGHT
)

self.settings_btn = ctk.CTkButton(
    self,
    text="Settings",
    fg_color="transparent",
    command=lambda: self.show_menu("settings"),
    width=175,
    height=BUTTON_HEIGHT
)

# Store buttons in a dictionary for easy access
self.buttons = {
    "version_control": self.version_control_btn,
    "asset_gallery": self.asset_gallery_btn,
    "settings": self.settings_btn
}

# Username label on the right
self.username_label = ctk.CTkLabel(
    self,
    text=username,
    text_color="gray",
    font=('Source Code Pro', 14, 'bold')
)
```

The grid method is used to place each of the buttons and label onto the top bar - each being vertically padded with the same variable in order to ensure consistent height across the bar.

A count controlled `for` loop is used to grid all of the menu frames themselves onto the screen consistently - each covering the remainder of the screen which isn't taken up by the top bar.

The `"version_control"` argument is passed to the `show_menu` method after this in order to display it by default

```python
# Layout using grid
ypad = (25, 15)
self.version_control_btn.grid(row=0, column=0, padx=(20, 10), pady=ypad)
self.asset_gallery_btn.grid(row=0, column=1, padx=10, pady=ypad)
self.settings_btn.grid(row=0, column=2, padx=10, pady=ypad)
self.username_label.grid(row=0, column=3, padx=20, pady=ypad, sticky="e")

# Grid all frames in the same position
for frame in self.frames.values():
    frame.grid(row=1, column=0, columnspan=4, sticky="nsew")

# Show initial frame
self.show_menu("version_control")
```

The `show_menu` method iterates over each button in the dictionary and uses a boolean expression to determine whether it should be displayed using the `BUTTON_COLOUR` or `DARKER_BLUE` - depending on if the name of the widget matches that of the `menu_name` parameter.

```python
def show_menu(self, menu_name: str) -> None:
    """
    * Show the selected menu frame and update button colors.

    Paramaters:
        menu_name: The name of the menu to display
    """
    # Update button colors
    for name, button in self.buttons.items():
        button.configure(fg_color=BUTTON_COLOUR if name == menu_name else DARKER_BLUE)

    # Raise the selected frame
    if menu_name in self.frames:
            self.frames[menu_name].tkraise()
```

This top bar will be shown in screenshots of each of these dashboard menus later on.

Version Control

The version control menu consists of a new project section and an existing projects section. The existing projects section is going to take up most of the space on the screen so I will first focus on that.

As outlined in the design section, the existing projects (or "Your Projects") section consists of a grid of cards which hold the information for a project. These cards display the project name, version number, creation date and preview image. The preview image will not be included in this sprint because it relies on visual preview features being developed.

Similar to the other frame classes, the `fg_colour` attribute is set to this one upon initialisation. The `ProjectCard` class takes the `LIGHTER_BLUE` colour in order to distinguish each card from both each other and the `DARK_BLUE` background.

Next, the grid is split into multiple rows in order to arrange the different widgets around the project card.

```python
class ProjectCard(ctk.CTkFrame):
    def __init__(
        self,
        master: ctk.CTkFrame,
        title: str,
        version: str,
        created_date,
        **kwargs
    ) -> None:
        super().__init__(master, fg_color=LIGHTER_BLUE, **kwargs)

        # Configure grid weights
        self.grid_columnconfigure(0, weight=1)
        self.grid_rowconfigure(4, weight=1)

        # Project title and version
        self.title_frame = ctk.CTkFrame(self, fg_color="transparent")
        self.title_frame.grid_columnconfigure(1, weight=1)
```

Similar to the `username` variable from the `TopMenuFrame` class, the `title`, `version` and `created_date` will later be swapped out for database values, but currently get pulled from a hard coded dictionary which will be shown shortly.

```python
# Project title and version
self.title_frame = ctk.CTkFrame(self, fg_color="transparent")
self.title_frame.grid_columnconfigure(1, weight=1)

self.title_label = ctk.CTkLabel(
    self.title_frame,
    text=title,
    anchor="w"
)
self.version_label = ctk.CTkLabel(
    self.title_frame,
    text=f"v{version}",
    text_color="gray"
)

# Placeholder image frame
self.image_frame = ctk.CTkFrame(
    self,
    fg_color=IMAGE_COLOUR,
    height=200
)

# Created date
self.date_label = ctk.CTkLabel(
    self,
    text=f"Created: {created_date}",
    text_color="gray"
)

# Launch button
self.launch_btn = ctk.CTkButton(
    self,
    text="Launch Scene",
    fg_color=BUTTON_COLOUR,
    height=BUTTON_HEIGHT
)
```

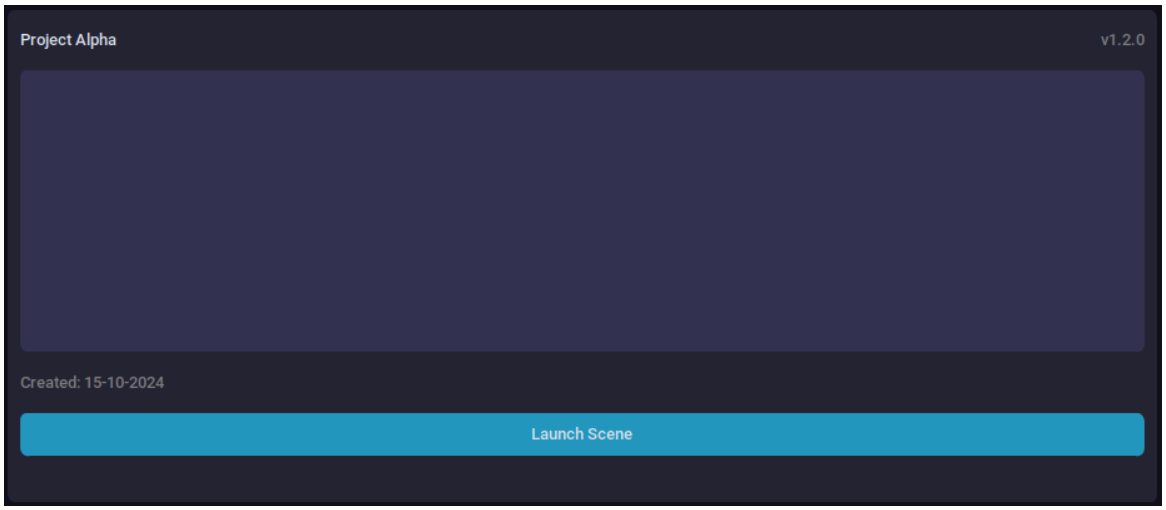
The `title_label` has no argument passed for the `text_color` parameter, resulting in the default white text colour. Conversely, the other pieces of text are created with the `"gray"` customtkinter colour. This is to draw attention to the project title as the most relevant piece of information when distinguishing projects.

The grid layout for these elements is fairly standard and similar to what has been explained in previous sections. The widgets are placed into their respective columns and rows, and the `sticky` arguments are used to choose if a widget should stick to the left or right hand side of the card (or span its whole width).

```python
# Layout
self.title_frame.grid(row=0, column=0, sticky="ew", padx=10, pady=(10, 5))
self.title_label.grid(row=0, column=0, sticky="w")
self.version_label.grid(row=0, column=1, sticky="e")

self.image_frame.grid(row=1, column=0, sticky="nsew", padx=10, pady=5)
self.date_label.grid(row=2, column=0, sticky="w", padx=10, pady=5)
self.launch_btn.grid(row=3, column=0, sticky="ew", padx=10, pady=(5, 10))
```

The `VersionControl` class is the main frame class which gets called for this menu by the `TopMenuFrame` class. It is responsible for placing the “Create Project” section widgets onto the screen, as well as the project cards recently discussed.

Similar to previous interface elements, these buttons are created using various constants, from the `UI` module, in order to ensure their consistency with the rest of the application.



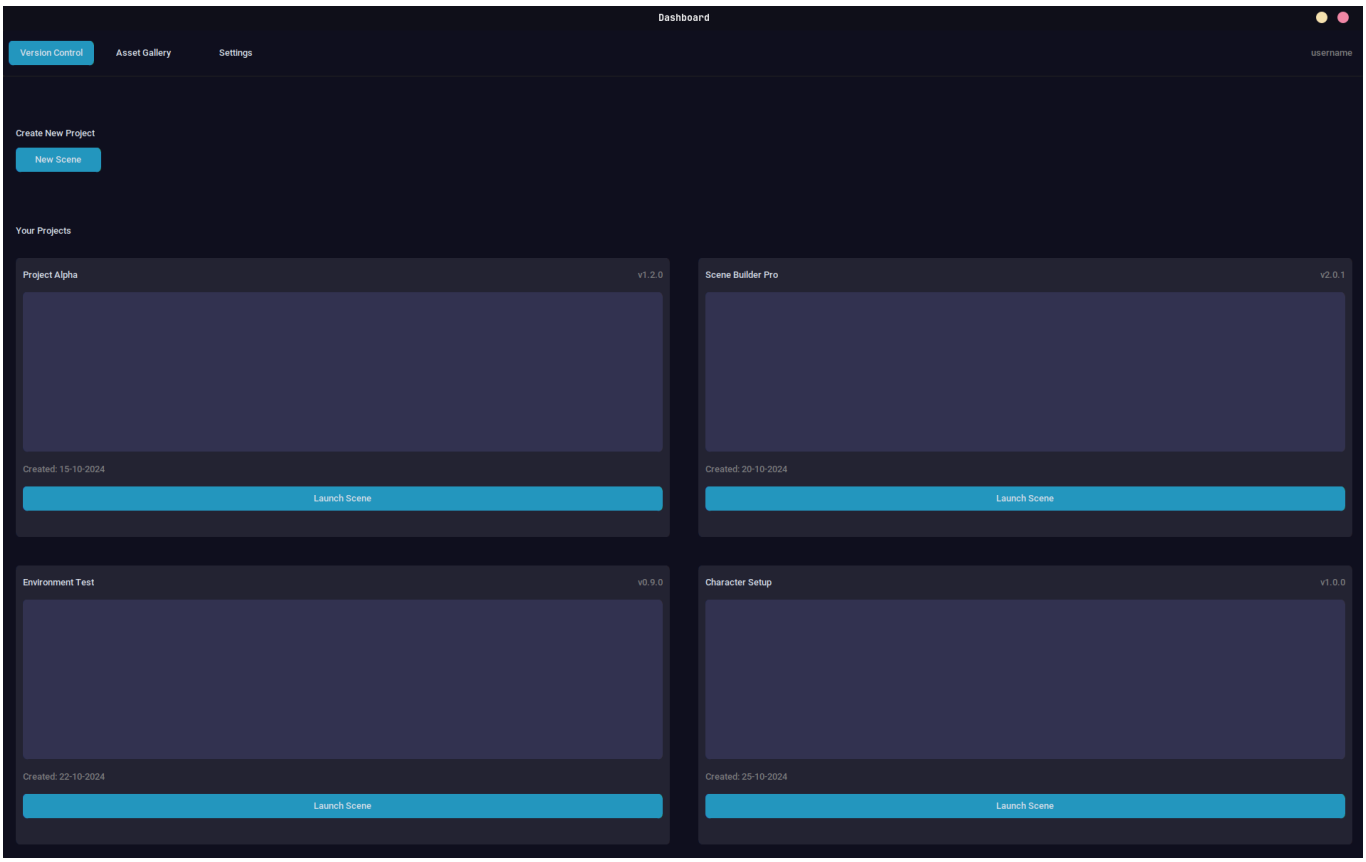

```python
class VersionControl(ctk.CTkFrame):
    def __init__(self, master, **kwargs):
        super().__init__(master, fg_color = DARK_BLUE, **kwargs)

        # Configure grid weights
        self.grid_columnconfigure((0, 1), weight=1)
        self.grid_rowconfigure((0, 1, 2), weight=1)

        # Create New Project section
        self.title_label = ctk.CTkLabel(
            self,
            text="Create New Project",
            anchor="center"
        )
        self.new_scene_btn = ctk.CTkButton(
            self,
            text="New Scene",
            fg_color=BUTTON_COLOUR,
            width=150,
            height=BUTTON_HEIGHT
        )

        # Project cards
        self.projects_label = ctk.CTkLabel(
            self,
            text="Your Projects",
            anchor="center"
        )
```

The `self.projects` list is a temporary variable which is only necessary for display purposes until the database features get implemented and linked into this menu.

```
self.projects = [
    ("Project Alpha", "1.2.0", "15-10-2024"),
    ("Scene Builder Pro", "2.0.1", "20-10-2024"),
    ("Environment Test", "0.9.0", "22-10-2024"),
    ("Character Setup", "1.0.0", "25-10-2024")
]

# Layout
self.title_label.grid(row=0, column=0, sticky="nw", padx=20, pady=(50, 10))
self.new_scene_btn.grid(row=0, column=0, sticky="nw", padx=20, pady=(75, 100))
self.projects_label.grid(row=0, column=0, sticky="w", padx=30, pady=(200, 0))

# Create and layout project cards
for i, (title, version, date) in enumerate(self.projects):
    row = (i // 2) + 1
    col = i % 2
    card = ProjectCard(
        self,
        title=title,
        version=version,
        created_date=date
    )
    card.grid(
        row=row,
        column=col,
        padx=(20, 30) if col else (30, 20),
        pady=20 if row else (0, 20),
        sticky="nsew"
    )
```

Generous padding is provided to some of the elements on this page so that the two different sections are properly separated.

As for the project cards, the data from each list element (later to be database values) are passed into the initialisation of a new `ProjectCard` object. These cards are then gridded by using boolean expressions, based on their place in the `self.projects` list, to assign their columns and rows.



Completed Test:

17.3	Navigation to version control - allows users to access the dashboard form the settings menu and gallery	Normal	user clicks on version control button	User redirected to version control	Dashboard

We now need to add functionality to the “New Scene” button.

1. `refresh_project_cards()`

```python
def refresh_project_cards(self):
    # Clear existing cards
    for widget in self.winfo_children():
        if isinstance(widget, ProjectCard):
            widget.destroy()

    # Create and layout project cards
    for i, (title, version, date) in enumerate(self.projects):
        row = (i // 2) + 1
        col = i % 2
        card = ProjectCard(
            self,
            title=title,
            version=version,
            created_date=date
        )
        card.grid(
            row=row,
            column=col,
            padx=(20, 30) if col else (30, 20),
            pady=20 if row else (0, 20),
            sticky="nsew"
        )
```

This function is designed with several key principles in mind:

- **Separation of concerns**: By extracting the card creation logic into a dedicated function, we maintain cleaner, more maintainable code.
- **Reusability**: The function can be called both during initialization and whenever the project list changes.
- **State consistency**: It first clears existing cards before recreating them, ensuring the UI always reflects the current state of the `projects` list.
- **Efficiency**: It only destroys `ProjectCard` instances, leaving other UI elements intact, which is more efficient than rebuilding the entire UI.

2. `open_add_project_dialog()`

This function employs a modal-like pattern that's appropriate for the customtkinter framework:

- **Modal-like behavior**: Creates an overlay that visually indicates the main application is inactive while the dialog is open.
- **Centered positioning**: Places the dialog in the center of the parent frame using relative coordinates for responsiveness.

```python
def open_add_project_dialog(self):
    # Create a semi-transparent overlay
    overlay = ctk.CTkFrame(self, fg_color=("gray80", "gray20"))
    overlay.place(relx=0.5, rely=0.5, anchor="center", relwidth=1, relheight=1)

    # Create the dialog frame
    self.dialog = AddProjectDialog(
        overlay,
        close_callback=lambda: self.close_dialog(overlay),
        create_callback=self.add_new_project,
        width=400,
        height=300,
        corner_radius=10,
        border_width=2,
        border_color="gray70"
    )
    self.dialog.place(relx=0.5, rely=0.5, anchor="center")
```

- **Separation from main UI**: By creating a new frame within an overlay, we ensure the dialog appears "on top" of the existing interface.
- **Callback architecture**: Passes callback functions to the dialog, allowing for loosely coupled component interaction.

```
def close_dialog(self, overlay):
    if self.dialog:
        self.dialog.destroy()
        self.dialog = None
    overlay.destroy()

def add_new_project(self, title, version):
    # Add the new project with current date
    current_date = datetime.now()
    self.projects.append((title, version, current_date))

    # Refresh the project cards display
    self.refresh_project_cards()
```

3. `close_dialog(overlay)`

This function follows good resource management practices:

- **Complete cleanup**: Removes both the dialog and its overlay to prevent memory leaks.
- **Null safety**: Checks if the dialog exists before attempting to destroy it.
- **Parameterization**: Takes the overlay as a parameter rather than storing it as an instance variable, reducing state management complexity.
- **Clear responsibility**: Has the single responsibility of cleaning up dialog resources.

4. `add_new_project(title, version)`

This function demonstrates proper data handling:

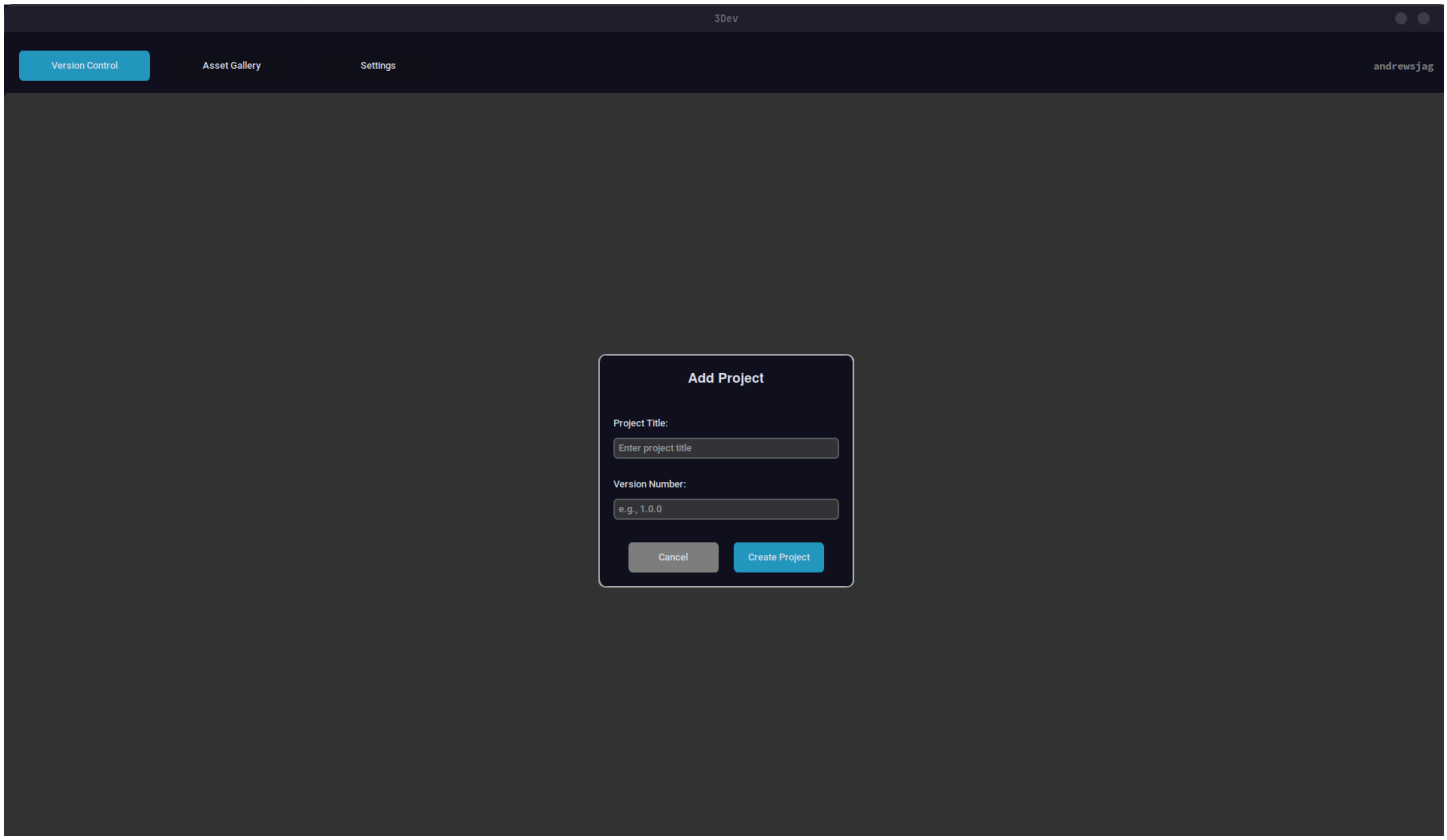
- **Data integrity**: Creates a new project entry with the correct structure (title, version, timestamp).
- **Current data**: Uses `datetime.now()` to timestamp new projects with the actual creation time.
- **UI synchronization**: Calls `refresh_project_cards()` to ensure the UI is updated to reflect the data change.
- **Parameter simplicity**: Takes only the user-provided information, handling the rest internally.

There are three main issues with this code right now:

➔ The dialogue opens on top of the current frame but covers it up with a gray background. We want this to be transparent so the rest of the frame is still visible behind the dialogue. One way I could do this is with a top level frame but that is less preferable and will only be a last resort as it means creating an entire new window every time we want to open a dialogue.

➔ The new project compresses the other projects in the grid in order to fit onto the page. This means we need a scrollable frame to keep all the elements fully visible.

➔ The added projects are currently not remembered when reopening the application. This will be solved in sprint 3 when the database integration is tackled



These are the changes I made to fix these issues:

Removed the gray overlay:

- The dialog now appears directly on top of the VersionControl frame without any intermediate overlay layer
- This allows the background content to remain fully visible

Dialog placement:

- Changed from relative positioning (`relx/rely`) to absolute positioning (`x/y`)
- Calculates the center position based on the parent's dimensions
- Includes fallback values (100px) in case the parent dimensions haven't been fully initialized

Simplified the close_dialog function:

- No longer needs an overlay parameter since there's no overlay to destroy
- Still properly cleans up the dialog when closed

Added lift() method call:

- Ensures the dialog appears on top of all other widgets in the parent frame
- This is important for proper visual layering when not using an overlay.

```python
def open_add_project_dialog(self):
    # Create the dialog frame directly on top of the current frame
    self.dialog = AddProjectDialog(
        self,  # Use self as the parent instead of an overlay
        close_callback=self.close_dialog,
        create_callback=self.add_new_project,
        width=400,
        height=300,
        corner_radius=10,
        border_width=2,
        border_color="gray70"
    )

    # Calculate the center position
    dialog_width = 400
    dialog_height = 300
    parent_width = self.winfo_width()
    parent_height = self.winfo_height()

    x_position = (parent_width - dialog_width) // 2 if parent_width > 0 else 100
    y_position = (parent_height - dialog_height) // 2 if parent_height > 0 else 100

    # Place the dialog in the center of the parent frame
    self.dialog.place(x=x_position, y=y_position, width=dialog_width, height=dialog_height)

    # Bring the dialog to the front
    self.dialog.lift()
```

This raised the following error at runtime:

raise ValueError("'width' and 'height' arguments must be passed to the constructor of the widget, not the place method")
ValueError: 'width' and 'height' arguments must be passed to the constructor of the widget, not the place method
Exception in Tkinter callback

Additionally, the dialogue is now just not visible when the new project button is pressed.

Here's how I fixed these issues:

```python
def open_add_project_dialog(self):
    # Define dialog dimensions
    dialog_width = 400
    dialog_height = 300

    # Get parent dimensions
    self.update_idletasks()  # Ensure geometry info is up to date
    parent_width = self.winfo_width()
    parent_height = self.winfo_height()

    # Calculate center position
    x_position = (parent_width - dialog_width) // 2 if parent_width > 0 else 100
    y_position = (parent_height - dialog_height) // 2 if parent_height > 0 else 100

    # Create the dialog frame with width and height in constructor
    self.dialog = AddProjectDialog(
        self,
        close_callback=self.close_dialog,
        create_callback=self.add_new_project,
        width=dialog_width,
        height=dialog_height,
        corner_radius=10,
        border_width=2,
        border_color="gray70"
    )

    # Place the dialog at the calculated position (without width/height)
    self.dialog.place(x=x_position, y=y_position)

    # Bring the dialog to the front
    self.dialog.lift()
```

Fixed the `place()` method usage:

- Removed the `width` and `height` arguments from the `place()` method
- These are now only specified in the constructor as required by customtkinter

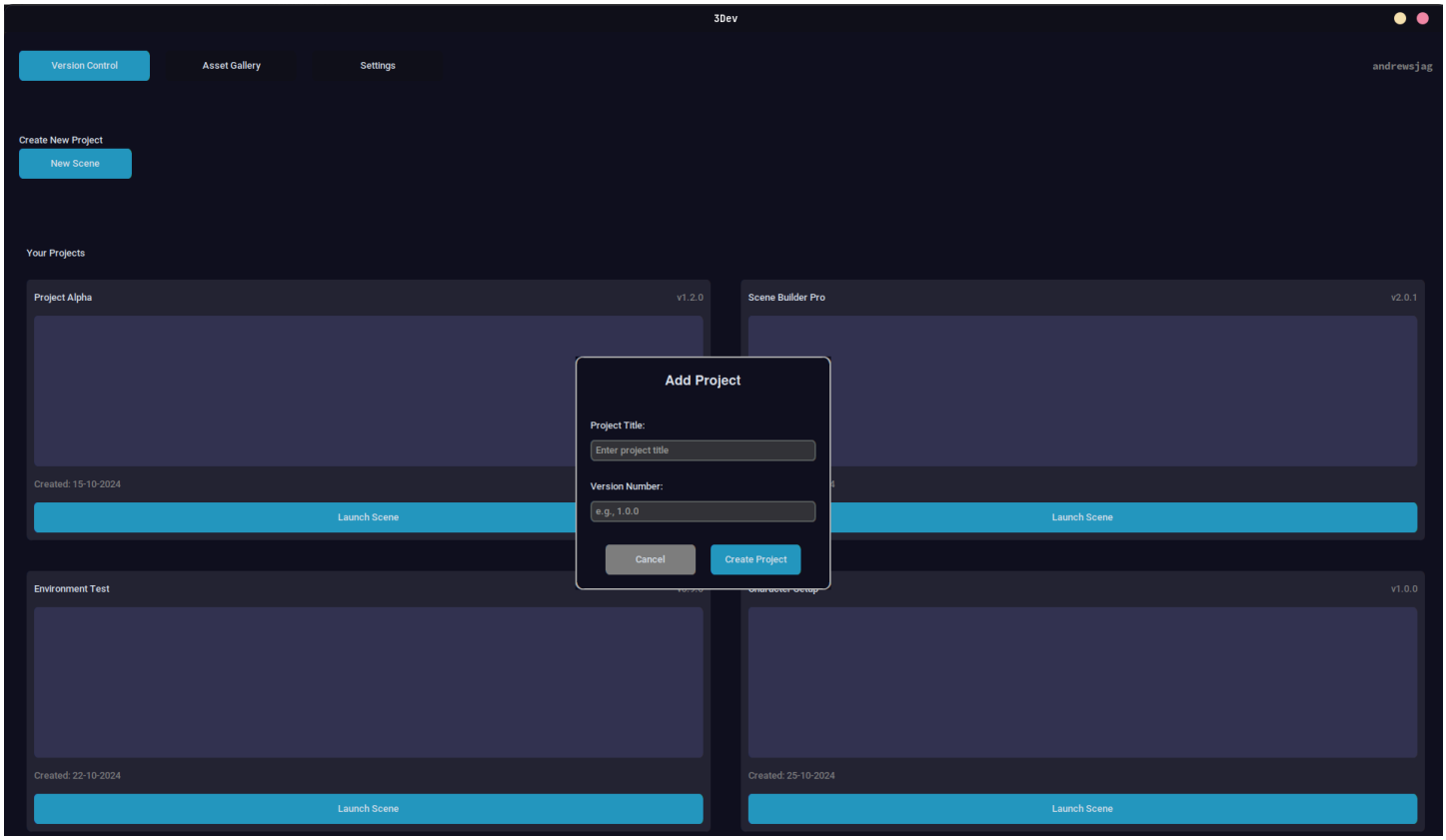
Added geometry update before getting parent dimensions:

- Added `self.update_idletasks()` before calculating positions

- This ensures the parent widget's geometry information is up to date before we use it

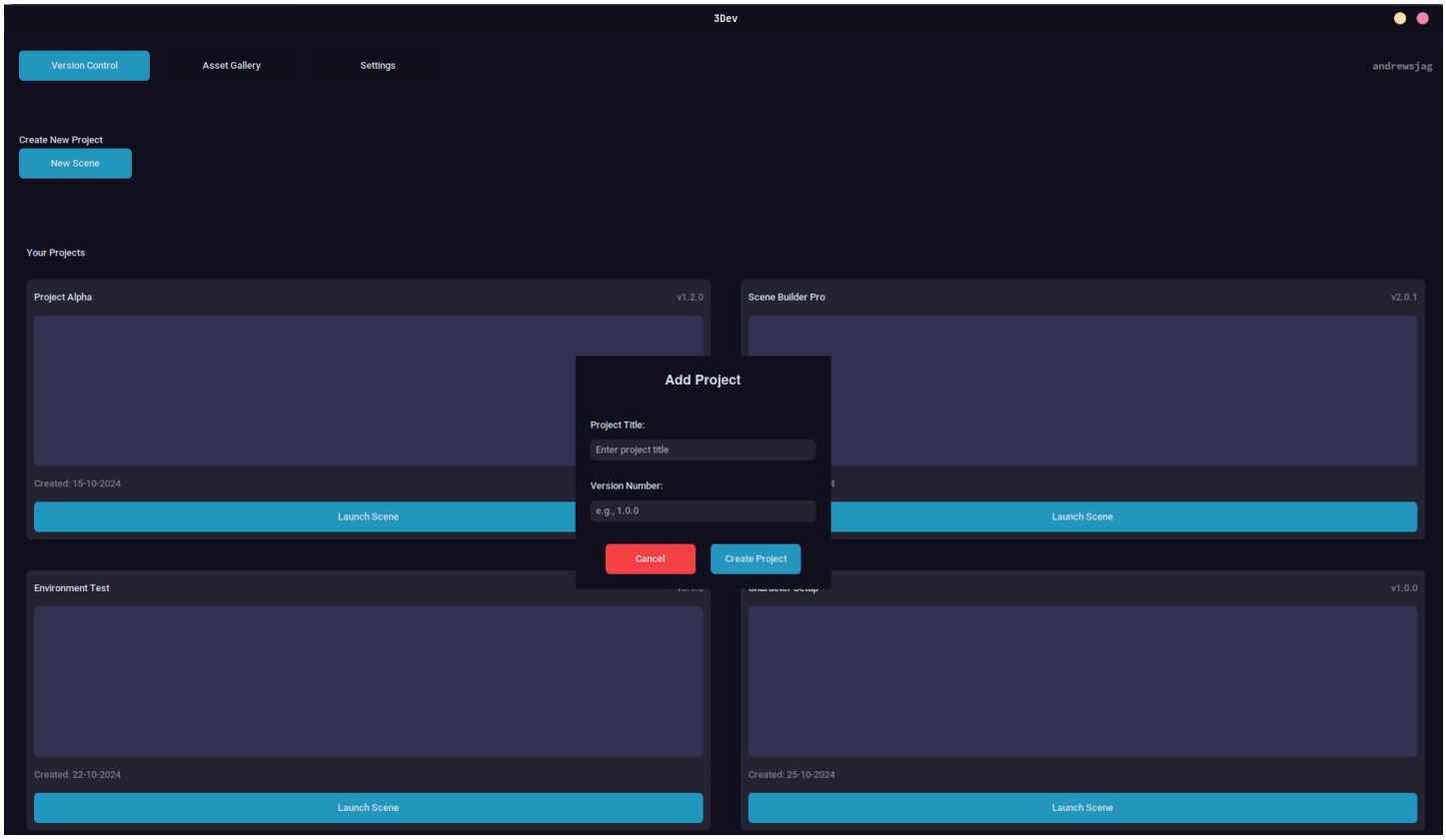
Simplified dialog positioning:

- The dialog is now created with the proper width and height
- Then it's placed at the calculated center position without setting dimensions again

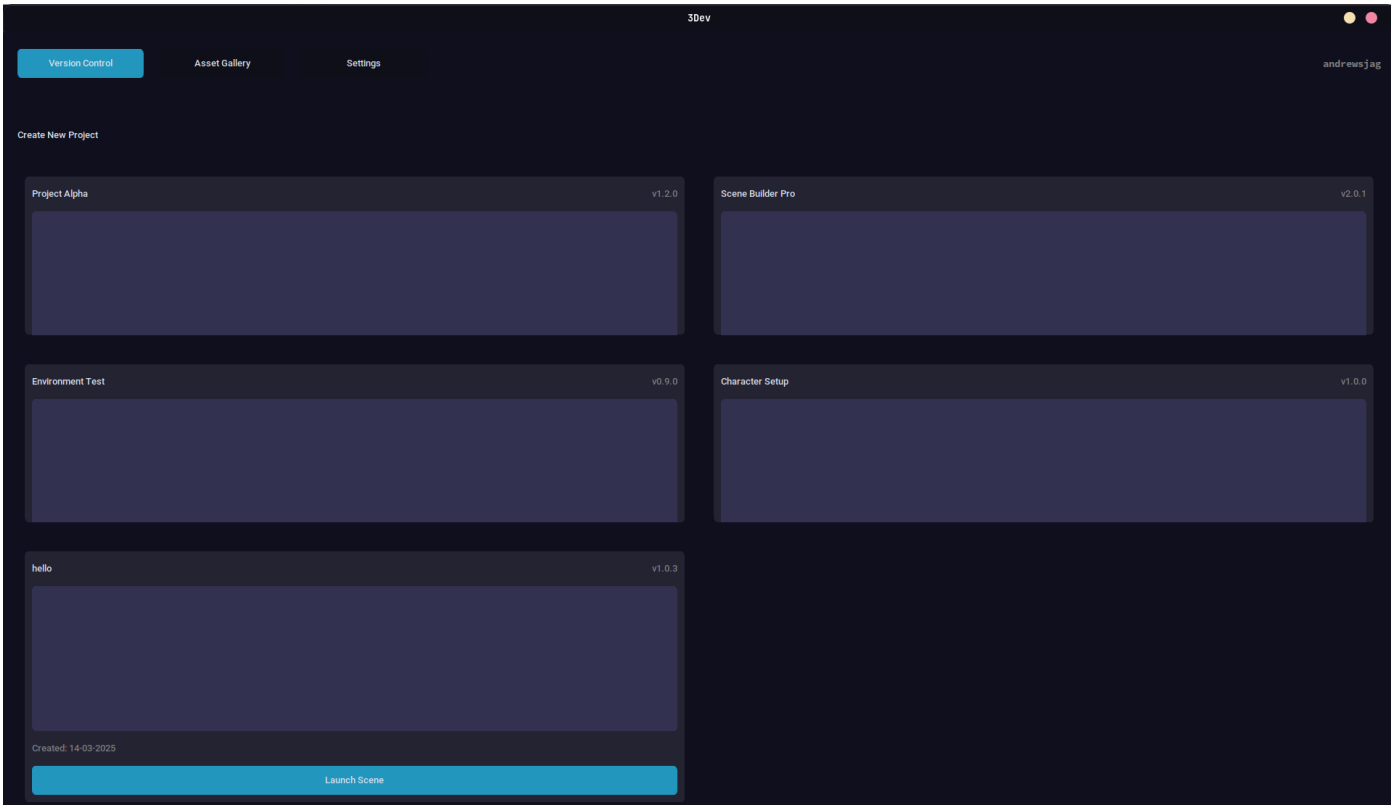


Now I just need to fix the styling of this frame to match the rest of the menu.

```
self.title_entry = ctk.CTkEntry(
    self,
    width=300,
    placeholder_text="Enter project title",
    fg_color=LIGHTER_BLUE,
    border_width=0
)
self.title_label = ctk.CTkLabel(
    self,
    text="Add Project",
    font=("Helvetica", 18, "bold")
)
self.cancel_btn = ctk.CTkButton(
    self,
    text="Cancel",
    fg_color=RED,
    command=self.close_callback,
    width=120,
    height=BUTTON_HEIGHT
)
```



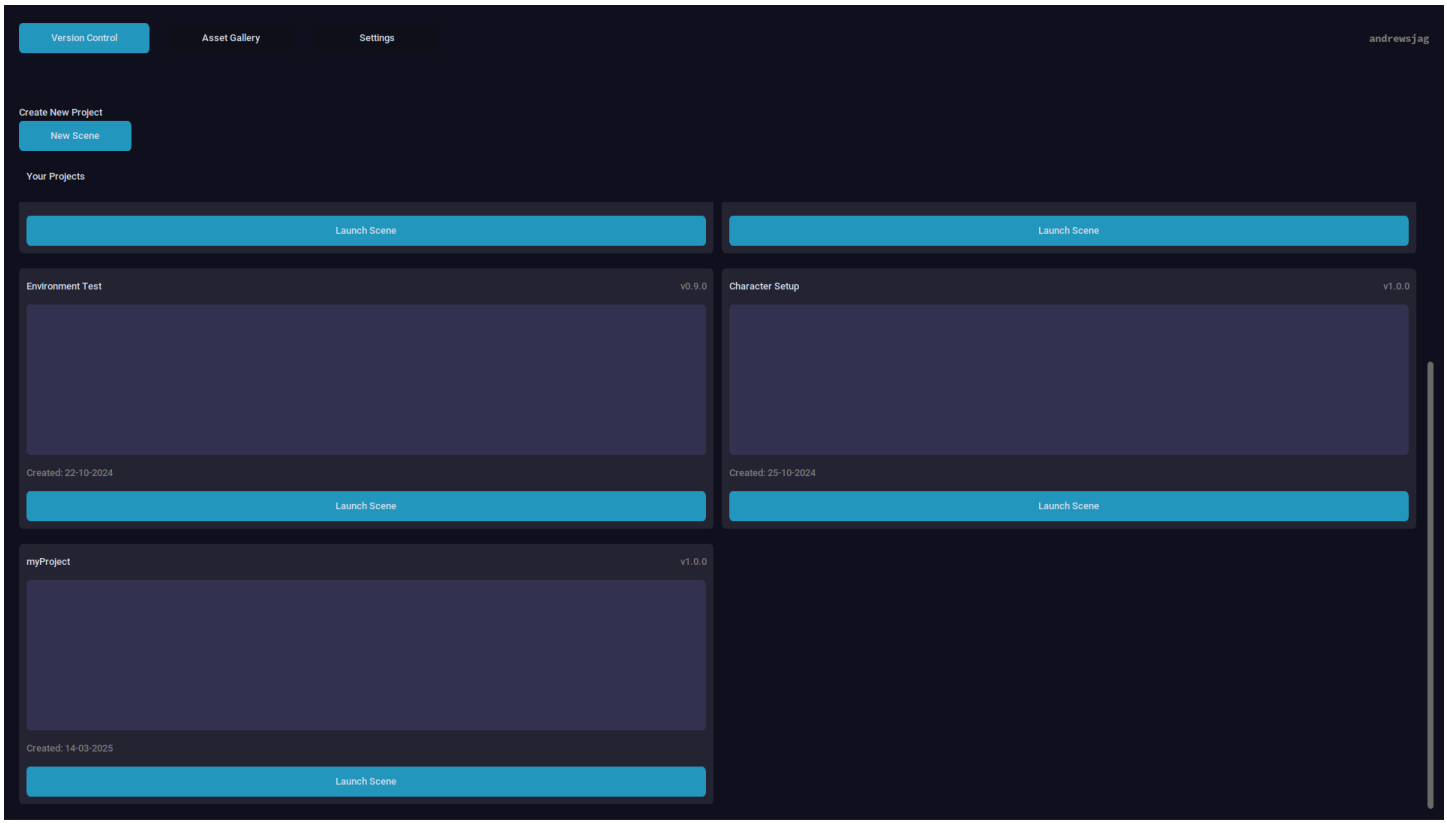
Now that these themes match, I need to fix the second issue of the card compression by implementing a scrollable frame.



Configuring the row weight to 1 (using `self.grid_rowconfigure()`) allows the scrollable frame to expand vertically instead of being limited to a minimum height.

```python
# Configure grid weights for main frame
self.grid_columnconfigure(0, weight=1)
self.grid_rowconfigure(2, weight=1)  # Make row with scrollable frame expandable

# Create a scrollable frame for projects
self.scrollable_frame = ctk.CTkScrollableFrame(
    self,
    fg_color="transparent",
    corner_radius=0
)

# Configure the scrollable frame's grid
self.scrollable_frame.grid_columnconfigure((0, 1), weight=1)

# Add the scrollable frame to the main layout
self.scrollable_frame.grid(row=2, column=0, sticky="nsew", padx=10, pady=10)
```

With these changes, the new project is now added below the existing ones with a scroll bar on the right hand side to navigate through these.



Asset Gallery

The `AssetCard` class is largely similar to the `ProjectCard` class so I will spend less time covering its explanation and design justifications.

The only change between these cards are the information they display and their size/position on the screen - however, the size/position aspect will be covered under the `AssetGallery` class explanation as it is this code in which the cards get gridded onto the screen.

```
class AssetCard(ctk.CTkFrame):
    def __init__(self, master, title: str, size: str, import_date: str, **kwargs):
        super().__init__(master, fg_color=LIGHTER_BLUE, **kwargs)

        # Configure grid weights
        self.grid_columnconfigure(0, weight=1)
        self.grid_rowconfigure(1, weight=1)

        # Header frame for title and size
        self.header_frame = ctk.CTkFrame(self, fg_color="transparent")
        self.header_frame.grid_columnconfigure(1, weight=1)

        self.title_label = ctk.CTkLabel(
            self.header_frame,
            text=title,
            anchor="w"
        )
        self.size_label = ctk.CTkLabel(
            self.header_frame,
            text=size,
            text_color="gray"
        )

        # Image placeholder
        self.image_frame = ctk.CTkFrame(
            self,
            fg_color=IMAGE_COLOUR,
            height=200
        )

        # Import date
        self.date_label = ctk.CTkLabel(
            self,
            text=f"Imported: {import_date}",
            text_color="gray"
        )

        # Layout
        self.header_frame.grid(row=0, column=0, sticky="ew", padx=10, pady=(10, 5))
        self.title_label.grid(row=0, column=0, sticky="w")
        self.size_label.grid(row=0, column=1, sticky="e")
        self.image_frame.grid(row=1, column=0, sticky="nsew", padx=10, pady=5)
        self.date_label.grid(row=2, column=0, sticky="w", padx=10, pady=(5, 10))
```

The search bar and some other widget groups for the asset gallery are placed within its own frame rather than the main content frame itself, this is to make its placement and positioning easier to manage.

Below this bar, we currently iterate over each of the texture/background categories and assign their visual attributes based on their position in the list. However, this is just to get all of the elements on the screen before we implement the logic to support it.

```python
class AssetGallery(ctk.CTkFrame):
    def __init__(self, master, **kwargs):
        super().__init__(master, fg_color=DARK_BLUE, **kwargs)

        # Configure grid weights
        self.grid_columnconfigure(0, weight=1)
        self.grid_rowconfigure(3, weight=1)  # Make asset grid expandable

        # Search bar
        self.search_frame = ctk.CTkFrame(self, fg_color="transparent")
        self.search_frame.grid_columnconfigure(0, weight=1)

        self.search_entry = ctk.CTkEntry(
            self.search_frame,
            placeholder_text="Search assets...",
            height=40,
            fg_color=LIGHTER_BLUE,
            border_width=0
        )

        # Category tabs
        self.category_frame = ctk.CTkFrame(self, fg_color="transparent")
        self.categories = ["Textures", "Backgrounds"]
        self.category_buttons: List[ctk.CTkButton] = []

        for i, category in enumerate(self.categories):
            btn = ctk.CTkButton(
                self.category_frame,
                text=category,
                fg_color=BUTTON_COLOUR if i == 0 else "transparent",
                height=30
            )
            btn.grid(row=0, column=i, padx=(0 if i == 0 else 10, 10), pady=10)
            self.category_buttons.append(btn)
```

In order to switch between the tab colours properly, I will add a command function for the category buttons both to call. Currently this will control just their appearance, but in future development the displayed cards will be filtered based on this user decision.

The rest of the UI elements are just more standard frames, labels and buttons.

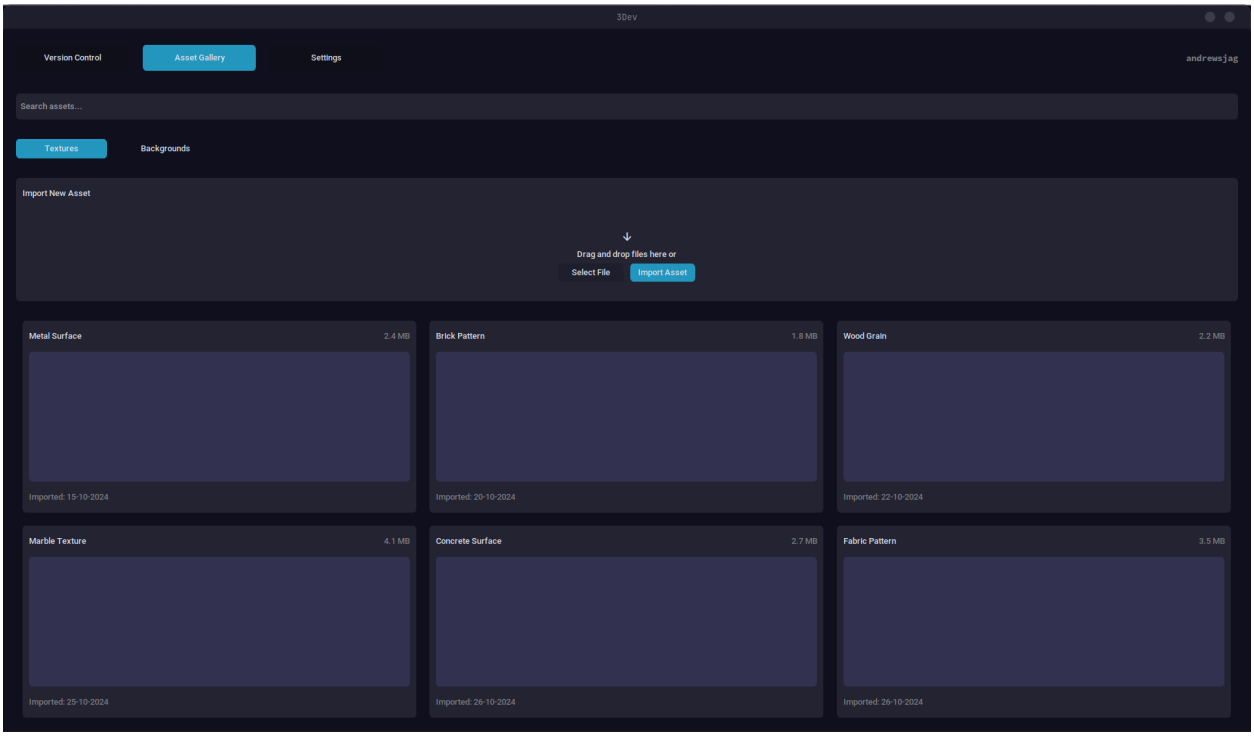
All of the import related interface is just visual for now, as is the rest of the interface, but this code will likely become considerably more complex once the functionality is implemented. This is something that I need to consider moving forward with allocating my time effectively - as drag and drop is not an essential feature (given the select functionality).

```python
# Category tabs
self.category_frame = ctk.CTkFrame(self, fg_color="transparent")
self.categories = ["Textures", "Backgrounds"]
self.category_buttons: List[ctk.CTkButton] = []

def category_button_command(index):
    for i, btn in enumerate(self.category_buttons):
        btn.configure(fg_color=BUTTON_COLOUR if i == index else "transparent")

for i, category in enumerate(self.categories):
    btn = ctk.CTkButton(
        self.category_frame,
        text=category,
        fg_color=BUTTON_COLOUR if i == 0 else "transparent",
        height=30,
        command=lambda index=i: category_button_command(index)
    )
    btn.grid(row=0, column=i, padx=(0 if i == 0 else 10, 10), pady=10)
    self.category_buttons.append(btn)
```

Just like the card class itself, the placement code for the asset cards resembles that of the project cards heavily. This may be an area of opportunity when it comes to refactoring code later on but for now this code is sufficient since there is no measurable performance impact by having the code optimally modularised. Additionally, the code is simple enough that the readability is not degraded much by the lack of modularity here.

```
# Import section
self.import_frame = ctk.CTkFrame(self, fg_color=LIGHTER_BLUE)
self.import_frame.grid_columnconfigure(0, weight=1)

self.import_label = ctk.CTkLabel(
    self.import_frame,
    text="Import New Asset",
    anchor="w"
)

self.drop_frame = ctk.CTkFrame(
    self.import_frame,
    fg_color=LIGHTER_BLUE,
    height=150
)
self.drop_frame.grid_columnconfigure(0, weight=1)
self.drop_frame.grid_rowconfigure(0, weight=1)

# Drop zone content
self.drop_icon_label = ctk.CTkLabel(
    self.drop_frame,
    text="↓",  # Simple arrow as placeholder for icon
    font=("Source Code Pro", 24)
)
self.drop_text_label = ctk.CTkLabel(
    self.drop_frame,
    text="Drag and drop files here or"
)

# Import buttons
self.button_frame = ctk.CTkFrame(
    self.drop_frame,
    fg_color="transparent"
)

self.select_btn = ctk.CTkButton(
    self.button_frame,
    text="Select File",
    fg_color="#1e222e",
    width=100
)
self.import_btn = ctk.CTkButton(
    self.button_frame,
    text="Import Asset",
    fg_color=BUTTON_COLOUR,
    width=100
)
```

```
# Asset grid
self.asset_grid = ctk.CTkFrame(self, fg_color="transparent")
self.asset_grid.grid_columnconfigure((0, 1, 2), weight=1)

# Sample assets
self.assets = [
    ("Metal Surface", "2.4 MB", "15-10-2024"),
    ("Brick Pattern", "1.8 MB", "20-10-2024"),
    ("Wood Grain", "2.2 MB", "22-10-2024"),
    ("Marble Texture", "4.1 MB", "25-10-2024"),
    ("Concrete Surface", "2.7 MB", "26-10-2024"),
    ("Fabric Pattern", "3.5 MB", "26-10-2024")
]

# Layout
# Search bar
self.search_frame.grid(row=0, column=0, sticky="ew", padx=20, pady=(20, 10))
self.search_entry.grid(row=0, column=0, sticky="ew")

# Category tabs
self.category_frame.grid(row=1, column=0, sticky="w", padx=20, pady=10)

# Import section
self.import_frame.grid(row=2, column=0, sticky="ew", padx=20, pady=10)
self.import_label.grid(row=0, column=0, sticky="w", padx=10, pady=(10, 5))
self.drop_frame.grid(row=1, column=0, sticky="ew", padx=10, pady=10)

# Drop zone content layout
self.drop_icon_label.grid(row=0, column=0, pady=(20, 0))
self.drop_text_label.grid(row=1, column=0)
self.button_frame.grid(row=2, column=0, pady=(0, 20))
self.select_btn.grid(row=0, column=0, padx=5)
self.import_btn.grid(row=0, column=1, padx=5)

# Asset grid
self.asset_grid.grid(row=3, column=0, sticky="nsew", padx=20, pady=10)

# Create and layout asset cards
for i, (title, size, date) in enumerate(self.assets):
    row = i // 3
    col = i % 3
    card = AssetCard(
        self.asset_grid,
        title=title,
        size=size,
        import_date=date
    )
    card.grid(row=row, column=col, padx=10, pady=10, sticky="nsew")
```



Completed Test:

17.1	Navigation to gallery - allows users to access the asset gallery from the main dashboard and settings menu	Normal	user clicks on gallery button	User redirected to asset gallery	Dashboard

Settings

The settings page is split into 3 sections - User Preferences, Render Preferences and Security. The original designs for this page had more sections and therefore included a side panel on which the user could click different section headers to skip to that section. These sections, and the settings included in them, were chosen purely for concept display purposes. Now, after the proper consideration of required settings, I have realised that the true length of this menu is not as long as once thought. As a result of this I have decided to remove the sidebar - given that it is largely unnecessary for the additional time and effort it would require to implement.

So that I could test this new design idea before moving forward to database linking work, I quickly rushed a version of the menu in order to troubleshoot possible design faults. I am going to reference relevant code throughout this process but will wait until I have a working interface design before going through all of the code, explaining its functionality and justifying its design.

My first design displays the three different settings sections slightly separated, with each individual setting displayed with the same size.

The main issue with this design is the sizing, particularly of the slider widgets. Since customtkinter scales using heights and widths, making all widgets the same scale means that the diameter of the slider equals the height of the buttons/text boxes. Because sliders hold no text based information, they do not need to be as tall as the other widgets, making them look largely out of place in the current design.

```
# Image width slider
width_slider = ctk.CTkSlider(
    self.render_prefs,
    from_=480,
    to=3840,
    number_of_steps=14,
    height=40,  # Increased height
    width=250  # Increased width
)
self.render_prefs.add_setting("Image Width", width_slider)
```

However, this design separates the distinct settings sections and clearly labels each of the individual settings using a label which is sufficiently padded to the left of the given setting.

In order to fix the visual issue of the last design iteration, I changed the `height` attribute of the slider widgets from 40 to 16.

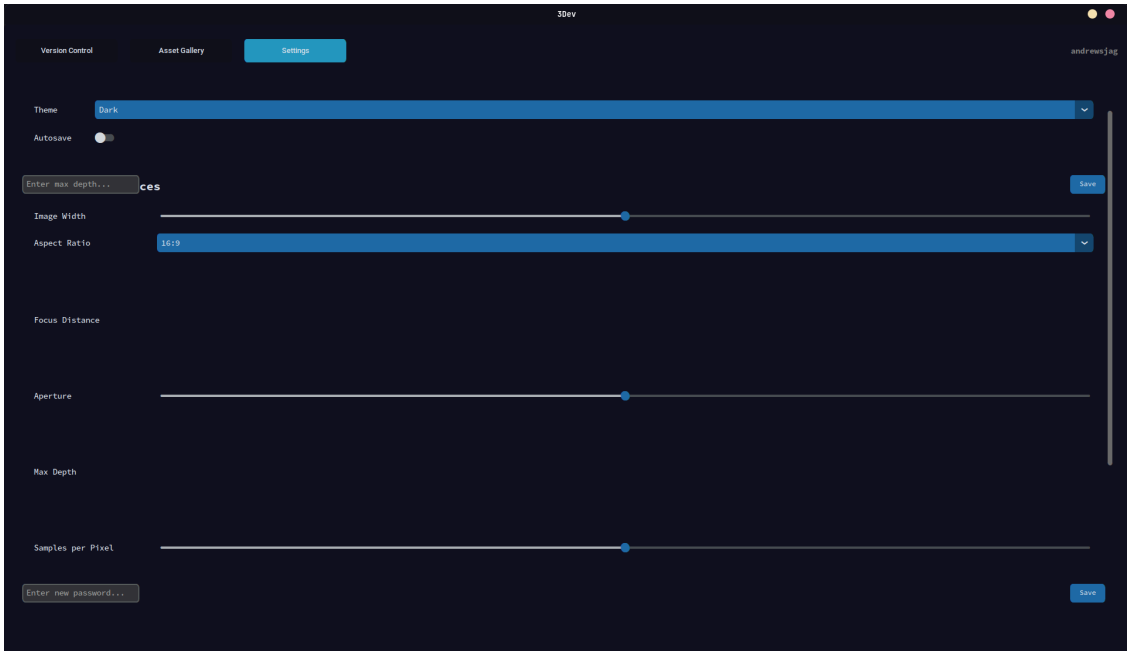
Additionally, I was previously writing the code for each setting individually - resulting in repetition which harms readability and wastes development time (especially if I decided to continue adding more setting options in the future). To combat this, I have decided to create a `SettingsSection` class responsible for modularising the code for each section - giving its own frame and own grid to house its settings widgets.

```python
class SettingsSection(ctk.CTkFrame):
    def __init__(self, master, title: str, **kwargs):
        super().__init__(master, fg_color=DARK_BLUE, **kwargs)

        self.grid_columnconfigure(1, weight=1)

        self.title = ctk.CTkLabel(
            self,
            text=title,
            font=("Source Code Pro", 20, "bold"),
            anchor="w"
        )
        self.title.grid(row=0, column=0, columnspan=2, sticky="W", padx=20, pady=(20, 15))

        self.current_row = 1
```

This class includes a method that generalises the creation of a setting option, including its label and variable widget. The widgets are added and gridded according to the required types which are passed as arguments into the method. Each of these widgets are slaves to their respective `SettingsSection` frame, rather than the main settings content frame.

This function has caused a logic error, resulting in the different settings options not lining up with their labels and also having text boxes fitting into the wrong column - vastly reducing their width and covering up other labels.

```python
def add_setting(self, label: str, widget: ctk.CTkBaseClass, button: ctk.CTkButton = None) -> None:
    """Add a setting row with a label, control widget, and optional button."""
    label = ctk.CTkLabel(
        self,
        text=label,
        anchor="w",
        font=("Source Code Pro", 14)
    )
    label.grid(row=self.current_row, column=0, sticky="w", padx=20, pady=8)

    if button:
        # Create a frame to hold the widget and button
        container = ctk.CTkFrame(self, fg_color="transparent")
        container.grid(row=self.current_row, column=1, sticky="ew", padx=20, pady=8)
        container.grid_columnconfigure(1, weight=0)  # Don't expand the button

        widget.grid(row=0, column=0, sticky="ew", padx=(0, 10))
        button.grid(row=0, column=1, sticky="e")

    else:
        widget.grid(row=self.current_row, column=1, sticky="ew", padx=20, pady=8)

    self.current_row += 1
```



I have identified a few key mistakes in my previous code, some of which involve the `add_setting` method and some originating from the `SettingsSection` class itself. There are a few things I need to do in order to properly fix this class and optimise it for effective modularity.

- ➔ Grid Column Configuration:
  - Add properly configured three columns in `SettingsSection` class:
    - **Column 0:** Fixed width for labels
    - **Column 1:** Expandable for widgets (`weight=1`)
    - **Column 2:** Fixed width for save buttons (`weight=0`)
  - Ensures consistent alignment of all elements across the different settings sections without having to grid elements consistently by hand. This reduces the risk of gridding mistakes (like the ones which caused the previous output) by handling it modularly.

- ➔ Simplify Widget Layout:
  - I am going to move away from the container frame approach (like the one used in the `AssetGallery` class) and replace it with direct grid positioning.
  - This fixes the issue of misaligned widgets and scattered labels/entry boxes as each row now has a consistent, and replicable, layout:
    - **label → variable widget → optional button**

- ➔ Section Spacing:
  - I will maintain consistent padding between sections (25px vertical) and between settings (20px horizontal and 8px vertical)
  - This cements the visual hierarchy of the menu and the separation between sections, just like the first design achieved.

- ➔ ‘Save’ Button Positioning:
  - Save buttons must be properly aligned in their own column and consistently padded to make sure they are equally spaced from their respective input field
  - Again, effectively implementing this in the `SettingsSection` class means that all instances of this type of setting remain consistent with one another.

```python
class SettingsSection(ctk.CTkFrame):
    def __init__(self, master, title: str, **kwargs):
        super().__init__(master, fg_color=DARK_BLUE, **kwargs)

        # Configure grid weights - critical for proper label/widget alignment
        self.grid_columnconfigure(1, weight=1)  # Make widget column expandable
        self.grid_columnconfigure(2, weight=0)  # Fixed width for save buttons

        # Section title
        self.title = ctk.CTkLabel(
            self,
            text=title,
            font=("Source Code Pro", 20, "bold"),
            anchor="w"
        )
        self.title.grid(row=0, column=0, columnspan=3, sticky="w", padx=20, pady=(20, 15))

        self.current_row = 1

    def add_setting(self, label: str, widget: ctk.CTkBaseClass, button: ctk.CTkButton = None) -> None:
        """Add a setting row with a label, control widget, and optional button."""
        # Create and position the label
        label = ctk.CTkLabel(
            self,
            text=label,
            anchor="w",
            font=("Source Code Pro", 14)
        )
        label.grid(row=self.current_row, column=0, sticky="w", padx=20, pady=8)

        # Position the widget
        widget.grid(row=self.current_row, column=1, sticky="ew", padx=20, pady=8)

        # Add save button if provided
        if button:
            button.grid(row=self.current_row, column=2, sticky="e", padx=(0, 20), pady=8)

        self.current_row += 1
```

The `Settings` frame class itself is what houses all of the `SettingsSection` frames. This is a scrollable frame because the settings page was anticipated to be longer than it is. I have decided to keep it as a scrollable frame in case more settings are added later because the scroll bar takes almost no space away from the widgets themselves.

All of the variable settings widgets call upon methods, such as `self.change_theme` for the `theme_dropdown`, which have not yet been implemented with functionality but will later connect these widgets to the database.

```python
class Settings(ctk.CTkFrame):
    def __init__(self, master, **kwargs):
        super().__init__(master, fg_color=DARK_BLUE, **kwargs)

        self.grid_columnconfigure(0, weight=1)
        self.grid_rowconfigure(0, weight=1)

        # Create scrollable frame
        self.scrollable_frame = ctk.CTkScrollableFrame(
            self,
            fg_color="transparent"
        )
        self.scrollable_frame.grid(row=0, column=0, sticky="nsew", padx=25, pady=25)
        self.scrollable_frame.grid_columnconfigure(0, weight=1)
```

```python
# User Preferences Section
self.user_prefs = SettingsSection(self.scrollable_frame, "User Preferences")
self.user_prefs.grid(row=0, column=0, sticky="ew", pady=(0, 25))

# Theme dropdown with implementation
self.theme_dropdown = ctk.CTkOptionMenu(
    self.user_prefs,
    values=["Dark", "Light", "System"],
    width=200,
    height=32,
    font=("Source Code Pro", 14),
    command=self.change_theme
)
current_theme = "System" # Default to system
self.user_prefs.add_setting("Theme", self.theme_dropdown)

# Autosave switch
self.autosave_switch = ctk.CTkSwitch(  # Make autosave_switch an instance variable
    self.user_prefs,
    text="",
    height=32,
    width=60,
    command=self.toggle_autosave # Add command for autosave switch
)
autosave_selected = False # Default to deselect
self.user_prefs.add_setting("Autosave", self.autosave_switch)
```

The rest of the code from here is a repeat of this format for the remaining two sections and their respective settings - making use of the `SettingsSection` class to create a section object and adding each widget to it using the `add_setting` method.

Completed Tests:

17.2	Navigation to settings menu - allows users to navigate to the settings menu from the main dashboard and gallery	Normal	user clicks on settings button	User redirected to settings page	Dashboard

(As can be seen on the right hand side of the page, the window is scrollable, however there are not enough settings present for this to have any effect).

17.4	Scrollable Settings - allows user to view all settings easily	Normal	user drags scroll bar	Page scrolls up/down	Dashboard

Testing

Personal Evaluation:

Due to time constraints, I am going to end the programming of this sprint here and complete the database connection during sprint 3.

As a result of this decision, some planned tests for this sprint are currently in a failed state - so will be completed in the next sprint. I will include an updated test plan at the beginning of the testing section which covers the completed features so far.

Because lots of requirements and tests are being moved around between sprints, the testID for each test is not likely to match up properly with the corresponding success criteria. However, the last column in the test records indicates the success criteria it links to.

As mentioned above, the following tests are all currently failed due to incompletion at this time. These tests are being rolled over into sprint 3 in order to remedy this:

9.11	Login success - allows users to login to an existing account.	Normal	username: validUserName password: correctPassword	User successfully authenticated, Success message: "Login success"	Multiple Users
9.12	Login fail - invalid username - prevents users from logging into a non existing account	Erroneous	username: invalidUserName password: anything	Error message: "username not found"	Multiple Users
9.13	Login fail - incorrect password - prevents users from logging into someone else's account	Erroneous	username: validUserName password: incorrectPassword	Error message: "Password Incorrect"	Multiple Users

9.22	Signup fail - username taken - ensures unique usernames for all accounts	Erroneous	username: existingUsername password: validPassword	Error message: "Username taken"	Multiple Users

9.3	Login button redirection - sends user into software once credentials are validated	Normal	Click login button (with valid credentials)	User redirected to dashboard	Multiple Users

The failure of the above tests in this sprint mean that the user credential validation and user database adding success criteria have not yet been fulfilled. This is going to be remedied by the implementation of database integration during the next sprint.

11.1	Import fail - incorrect file format - prevents users importing unsupported files as assets	Erroneous	Select invalid file format, file.invalid, to import	Error message: "Invalid format - file extension {invalid} not supported"	Imports
11.2	Import success - Ensures imported files are being added to database	Normal	Select file to import and apply it to a scene	Image rendered correctly with new asset applied	Imports

11.3	Display assets by category - Ensures assets are being assigned the correct categories on saving	Normal	Save multiple assets with varying categories and output the asset table content	Assets should be grouped into textures and backgrounds as well as divided into categories under each of those	Imports

The failure of the above tests in this sprint means that the import validation and saving success criteria have not been fulfilled yet. This will also aim to be remedied in sprint 3 by implementing database integration to the interface.

The following tests cover the requirements completed in this sprint:

9.14	Login fail - empty fields - stops users logging in with empty fields	No Input	username: empty password: empty	Error message: "Please fill login fields"	Multiple Users
9.21	Signup fail - invalid pass length - increases security	Erroneous	username: validUsername password: 123	Error message: "password must be at least 6 characters"	Multiple Users
9.22	Signup fail - invalid username (special characters) - increases security	Erroneous	username: invalidUsername password: validPassword Conf: =password	Error message: "Username cannot contain special characters"	Multiple Users
9.23	Signup fail - invalid pass (not enough special characters) - increases security	Erroneous	username: validUsername password: 987hfaii Conf: =password	Error message: "password must be at least 2 special characters"	Multiple Users
9.24	Signup fail - invalid pass (>3 consecutive digits) - increases security	Erroneous	username: validUsername password: 1234mnb!! Conf: =password	Error message: "password must have no more than 3 consecutive digits"	Multiple Users
9.25	Signup fail - passwords don't match - increases security and avoids user error	Erroneous	username: validUsername password: validPass Conf: != password	Error message: "passwords do not match"	Multiple Users

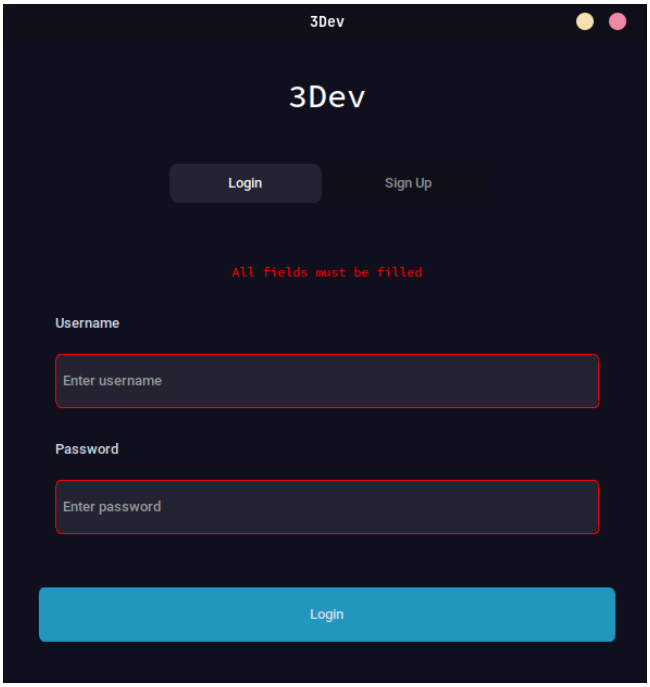
9.26	Signup fail - unfilled fields - stops users making accounts with blank properties	No Input	username: empty password: empty Conf: empty	Error message: "passwords do not match"	Multiple Users

The completion of the following tests were covered during the development of this sprint and fulfill the ‘Dashboard’ success criteria.

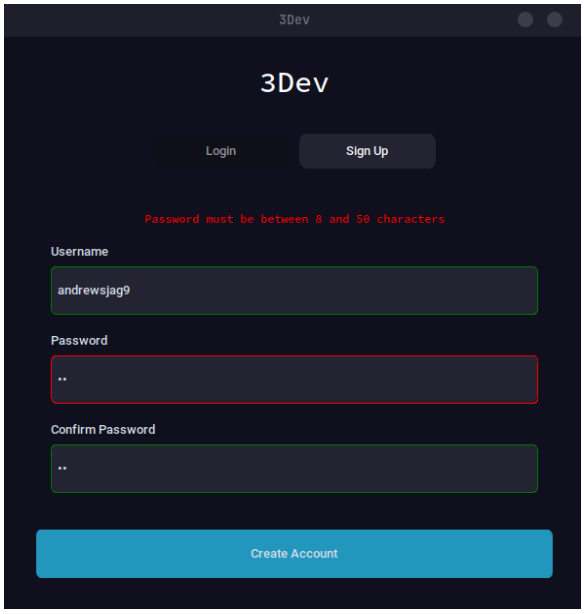
17.1	Navigation to gallery - allows users to access the asset gallery from the main dashboard and settings menu	Normal	user clicks on gallery button	User redirected to asset gallery	Dashboard
17.2	Navigation to settings menu - allows users to navigate to the settings menu from the main dashboard and gallery	Normal	user clicks on settings button	User redirected to settings page	Dashboard
17.3	Navigation to version control - allows users to access the dashboard form the settings menu and gallery	Normal	user clicks on version control button	User redirected to version control	Dashboard

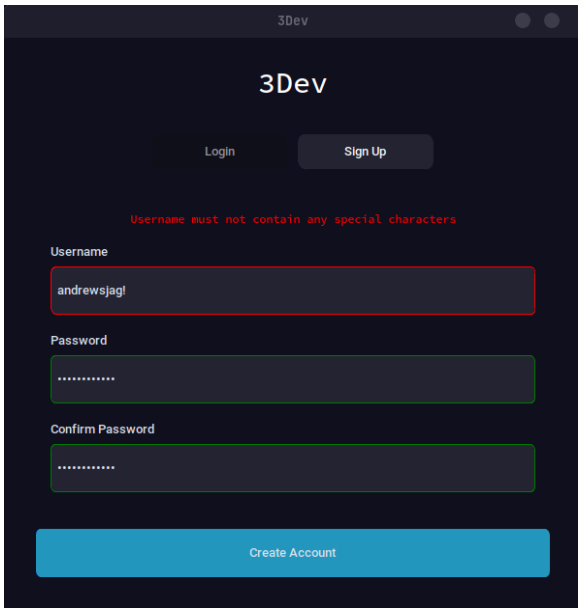
The following tests contribute to the 'Multiple Users' success criteria. More specifics about the success criteria met will be provided under the testID

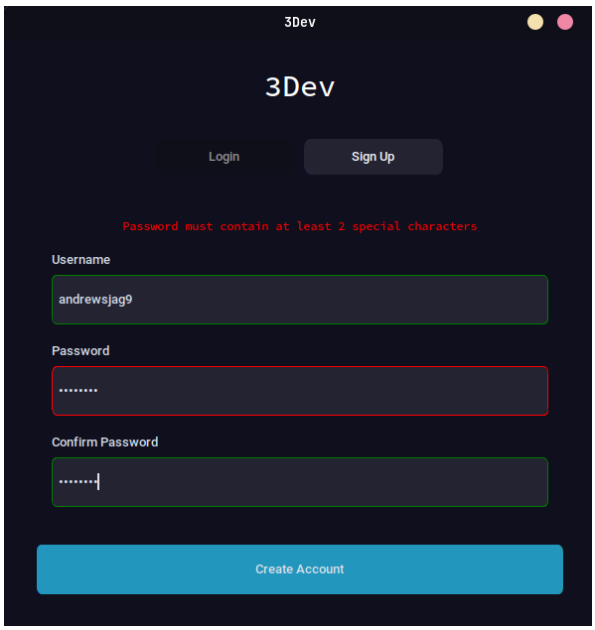
9.14:
Username/password fields for login and their validation

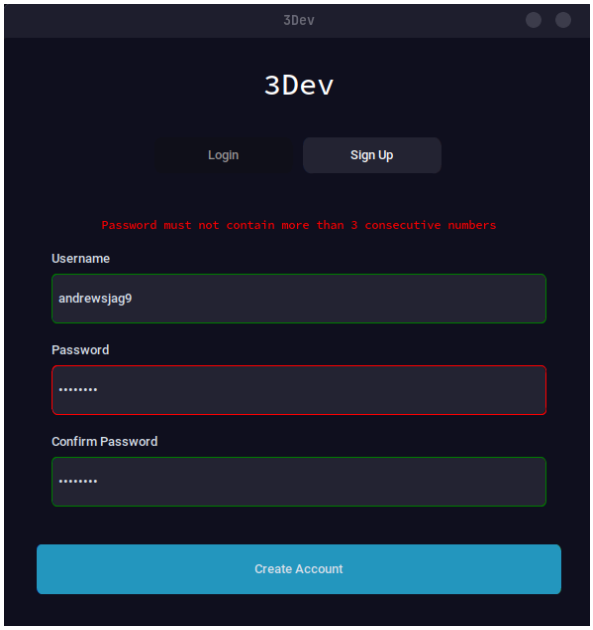


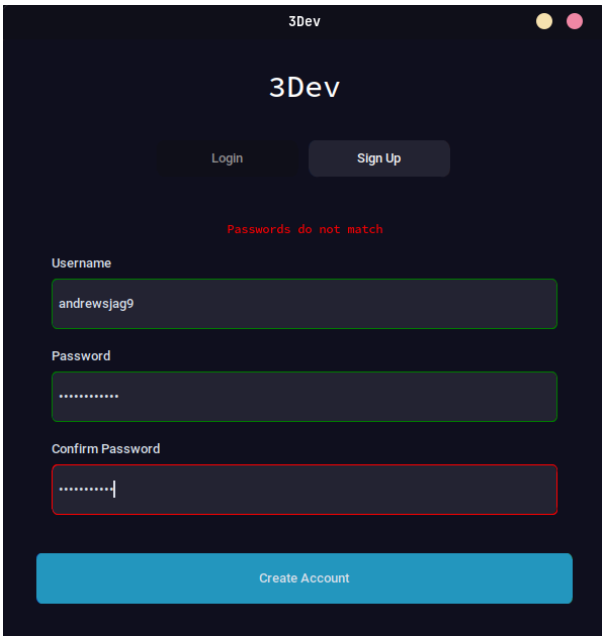
9.21, 9.22, 9.23, 9.24, 9.25, 92.6 (in order):
Sign up validation

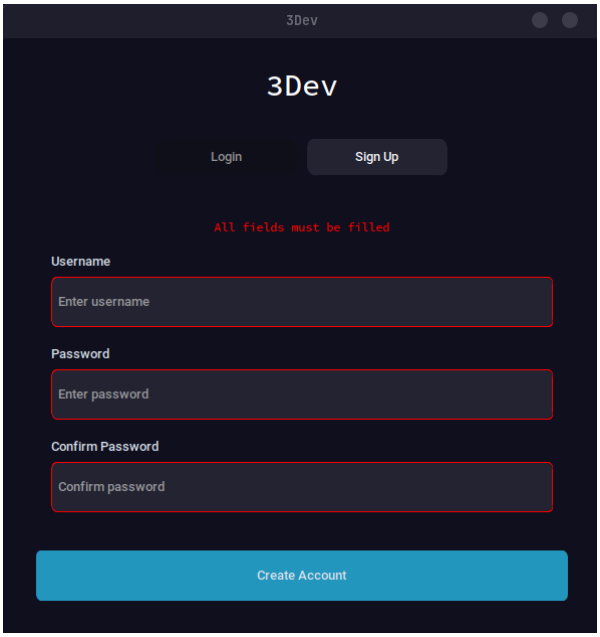












Sprint 3

Sprint Analysis

Stakeholder Input

I have asked both of my stakeholders to review the current state of my app (just the UI section since the ray tracer was not developed during sprint 2) after sprint 2. Previously, I have just asked Jason to review the interface, but this time I am also asking Martin with the hope of getting more mixed suggestions.

- ➔ *“The interface is nice to look at and easy to navigate, my main concern is that the majority of the UI is not functional and just visual”* - Martin

- ➔ *“Its nice and the theme fits well across the board, not much of it works or does anything right now though*” - Jason

Both stakeholders have expressed very similar views on the interface. The general conclusion is that the user interface is aesthetically pleasing but not functional at the moment so that should be my top priority.

This feedback sticks to the plan I had for sprint three at the end of sprint 2 (Integrating the database into the interface).

Due to time limitations, this means that quite a few requirements are not going to be met and a sizable portion of the project will not be completed. This is something that will be reviewed in the evaluation stage.

Detailed Success Criteria

1. User login/sign-up
    a. Valid login takes user to dashboard
    b. Valid Sign Up takes user to dashboard
    c. Valid Sign Up adds user to database

2. Asset Manager
    a. Imported assets listed alphabetically
    b. Search bar to look for assets (by title)
    c. Asset cards show from database

3. Settings Menu
    a. Altered settings save to database
    b. Logout button sends user back to sign-in page

4. Version Control Menu
    a. Scrollable menu with projects arranged in a grid of cards
    b. Project cards shown from database

| Requirement | Priority | Raised By | Description | Feature | Date Raised |

The following requirements have been stricken from the plan due to time constraints as explained above.

Object Manager	Must Have	Me (Jamie)	Tab with visual list of all the objects in the scene and their variable attributes on display to edit, as well as buttons to add and remove objects from the scene	Scene Creator	01/09/2024
Positionable Camera	Must Have	Me (Jamie)	Method for user to move the camera to different positions and orientations around a scene	Scene Creator	22/08/2024
Scene Templates	Could Have	Me (Jamie)	Introducing a few basic preset scenes for common use cases	Scene Creator	01/09/2024
Object Addition Menu	Must Have	Me (Jamie)	Pop up window when adding objects to chose their attributes, the quantity of the object you are adding and the position(s) of the object(s)	Scene Creator	05/09/2024
Backgrounds	Must Have	Martin	Having a range of different types of backgrounds which can be applied to a scene for different effects and use cases	Scene Creator	05/09/2024
Imports	Must Have	Martin	An option to import additional textures and backgrounds from your computer file system	Scene Creator	09/09/2024
Import Validity	Must Have	Martin, Me (Jamie)	Checks to ensure imported texture and background files are the correct format	Scene Creator	09/09/2024
Instances	Should Have	Me (Jamie)	Object transformations such as rotations and translations	Scene Creator	01/08/2024
Dark Mode	Should Have	Martin	Option to view the entire interface in a dark mode	Scene Creator	09/09/2024
Visual Scene Previews	Could Have	Me (Jamie)	Low detail preview of the scene as it is being created and edited	Scene Creator	05/09/2024
Illustrated Dropdowns	Could Have	Me (Jamie), Martin	Small diagrams next to dropdown option descriptions to illustrate the function of the option	Scene Creator	09/09/2024
Rendering Queue	Could Have	Me (Jamie)	If you want to render multiple scenes (or the same scene in various different ways), which might take some time you can create a queue of scenes to be rendered which will start one after another automatically	Scene Creator	05/09/2024

Parallel Rendering	Could Have	Me (Jamie)	If you want to render multiple scenes or versions of a scene at once, you can select a number of scenes to render in parallel with each other	Scene Creator	05/09/2024
Video Rendering	Won't Have	Me (Jamie)	The user will not be able to string together multiple versions of a scene and turn it into an animation	Scene Creator	05/09/2024
Real Time Rendering	Won't Have	Me (Jamie)	The scene won't be rendered in real time using ray tracing as it is being edited	Scene Creator	05/09/2024

The following requirements will be tackled in sprint 3.

Multiple Users	Must Have	Martin	Ability to sign in with different users who are working on different scenes	Version Control	01/09/2024
Savable Scenes	Must Have	Me (Jamie)	Allowing the user to save all the data for a scene (as opposed to just the final render)	Version Control	05/09/2024
Resolution Dropdown	Must Have	Me (Jamie)	Dropdown to edit the resolution of the rendered image (from a range of standard options) without changing the aspect ratio	Settings Menu	04/09/2024
Aspect Ratio Dropdown	Must Have	Jason	Dropdown to select aspect ratio of rendered image, from a range of standard options (including an option for a custom aspect ratio)	Settings Menu	04/09/2024
Texture/Background Manager	Should Have	Jason	Galleries showing all of your downloaded textures/backgrounds. Glass, steel & marble will be default materials available which are generated without using a texture map	Settings Menu	01/09/2024
Performance Toggles	Should Have	Me (Jamie)	Toggles to include or disclude effects such as shadows, metallic reflections, reflections, volumes, motion blur	Settings Menu	04/09/2024
Renders Slider	Should Have	Jason	Creating sliders to edit preferences such as depth of field, when rendering so values can be adjusted more intuitively	Settings Menu	05/09/2024
Saving Options	Should Have	Jason	Allow users to choose between overwriting the current file and creating a new file with the updated content when saving projects	Version Control	09/09/2024
Version Descriptions	Should Have	Me (Jamie), Martin	Allowing the user to add a short comment or description to each version of a project as they save it - similar to Git's commit messages	Version Control	09/09/2024

Gallery Searching	Should Have	Jason	Assigning multiple search terms to each texture/background to make searching more intuitive	Settings Menu	09/09/2024
Chronological Versions	Should Have	Me (Jamie), Martin	Showing the dates on which versions of a projet were saved and displaying them in reverse chronological order (top version most recent)	Version Control	10/09/2024

Test Plan

| ID | Description & Justification | Type | Required Input | Expected Output | Success Criteria |

The following tests have been stricken from the plan along with their requirements (explained above).

1.11	Navigation for object manager - allows the user to open an expanded view of their objects while editing a scene	Normal	Click expand tab button	Sidebar beam smoothly expands to full menu	Object Manager
1.12	Closing object manager - allows the user to have more screen space dedicated to viewing their scene	Normal	Click expand tab button	Sidebar menu smoothly collapses to a small beam on the side of the window	Object Manager
1.2	Object manger scene reactions - keeps the object manager constantly up to date with what is going on in the scene	Normal	Add object to scene	Object listing added to sidebar	Object Manager
1.3	Object manager dropdown - allows user to see more details about particular objects in the scene	Normal	Click dropdown arrow next to object listing	Object attributes expanded underneath	Object Manager
1.4	Object listing scene reactions - keeps object manager listing attributes up to date with what's happening in the scene	Normal	Change colour of object in scene	Object preview changes to correct colour	Object Manager
1.5	Open object attribute menu - shows user detailed display of object information	Normal	Double click on an object in the manager tab	Menu appears in the middle of screen showing attributes and larger preview	Object Manager
1.61	Sidebar tab switch (OM) - lets user navigate	Normal	Click object tab button	Sidebar switches from camera to	Object Manager

	sidebar		on sidebar	OM tab	
1.62	Sidebar tab switch (camera) - lets user navigate sidebar	Normal	Click camera tab button on sidebar	Sidebar switches from OM tab to camera tab	Object Manager, Positionable Camera

14.1	Object addition button - allows user to quickly add objects into the scene	Normal	User clicks addition button	Object addition menu opens in centre of screen	Object Addition Menu
14.2	Saving object success - allows scene to be updated with new objects	Normal	Click save object button (with valid attributes)	Object added to scene & object manager list	Object Addition Menu, Object Manager
14.31	Saving object fail - invalid field values - prevents user adding objects with invalid attributes	Erroneous	Click save object button (with invalid attributes)	Error message: "Invalid field value: {fieldData}", object not added to scene	Object Addition Menu
14.32	Saving object fail - empty fields - prevents user from adding objects without necessary attributes	No Input	Click save object button (with empty fields)	Error message: "Please fill in {fieldName} field", object not added to scene	Object Addition Menu

The following is the new test plan for sprint 3.

1.1	Successful sign up takes user to dashboard - ensures page redirection working correctly	Normal	Submit sign up with valid credentials	User redirected to dashboard and username displayed in top right	Multiple Users

1.2	Successful login takes user to dashboard - ensures sign-up adds user to database and user data is fetched correctly	Normal	Submit login with valid credentials for user created in test 1.1	User redirected to dashboard and username displayed in top right	Multiple Users

2.1	Asset cards presented alphabetically in the grid - makes it easier for user to find their assets without searching	Normal	Assets added to the database and user navigates to asset gallery	Asset cards listed alphabetically by asset name	Imports

2.21	Asset searching filters asset card display results	Normal	User types asset name into search bar (case insensitive)	Only asset with that name shown in gallery	Imports

2.21	Lack of asset sharing results in no filters - ensures users can see all their assets	No Input	User types nothing into search bar	OAll assets shown in gallery	Imports

2.31	Asset cards update with database (addition) - ensure asset gallery is accurate and up to date	Normal	Asset added to database	Asset cards shown in version control menu	Imports

2.32	Asset cards update with database (subtraction) - ensure asset gallery is accurate and up to date	Normal	Asset removed from database	Asset cards shown in version control menu	Imports

3.1	Altering settings changes database records - ensures settings are remembered and saved between logins	Normal	User edits one of the settings fields	Database field updated	Settings Menu

3.2	Logging out of account - lets users swap between accounts without having to restart the application	Normal	User clicks logout button	Redirected to sign-in page	Settings Menu

4.1	Version control menu scrollable - in order to view all projects on one menu page	Normal	User drags scroll bar down	Project cards grid scrolls down	Version Control

4.21	Project cards update with database (addition) - ensure version control menu is accurate and up to date	Normal	Project added to database	Project cards shown in version control menu	Version Control

4.22	Project cards update with database (subtraction) - ensure version control menu is accurate and up to date	Normal	Project removed from database	Project no longer shown in version control menu	Version Control

Programming

Data Manager

The `DataManager` class is responsible for managing the database - including creating, reading from, and writing to tables. The `init` and `connect` methods initialise the database before we start adding tables.

```
class DataManager:
    def __init__(self, db_file: str = "3Dev.db"):
        """Initialize database connection and create tables if they don't exist."""
        self.db_file = db_file
        self.conn = None
        self.create_tables()

    def connect(self):
        """Create a database connection with row factory enabled."""
        self.conn = sqlite3.connect(self.db_file)
        self.conn.row_factory = sqlite3.Row
        return self.conn
```

The Settings menu is split into three sections, each one of these sections is going to have its own dedicated table in order to keep non co-dependant data separate. Doing this avoids confusion and simplifies the code required to connect the relevant areas of the interface to the database.

Each table creation statement uses `CREATE TABLE IF NOT EXISTS`, which means it will only create the table if it isn't already present in the database. This prevents the duplication of tables which would lead to confusion and errors when fetching data.

The tables include timestamp fields (`created_at` and `updated_at`) to automatically track when records are created or modified - without the need for any user input.

After defining the SQL statements, I use the '`with`' context manager to ensure all database operations are performed within a single transaction. This maintains data integrity as either all operations succeed, or none of them do.

```python
def create_tables(self):
    """Create tables that mirror the settings menu structure."""
    # Users table - corresponds to Security section
    create_users_table = """
    CREATE TABLE IF NOT EXISTS users (
        user_id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE NOT NULL,
        password_hash TEXT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );
    """

    # User preferences - corresponds to User Preferences section
    create_user_preferences_table = """
    CREATE TABLE IF NOT EXISTS user_preferences (
        user_id INTEGER PRIMARY KEY,
        theme TEXT DEFAULT 'light',
        auto_save BOOLEAN DEFAULT 0,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (user_id) REFERENCES users (user_id)
            ON DELETE CASCADE
    );
    """

    # Render preferences - corresponds to Render Preferences section
    create_render_preferences_table = """
    CREATE TABLE IF NOT EXISTS render_preferences (
        user_id INTEGER PRIMARY KEY,
        render_name TEXT DEFAULT 'Default',
        image_width INTEGER DEFAULT 800,
        aspect_ratio REAL DEFAULT 1.778,
        focus_distance REAL DEFAULT 10.0,
        aperture REAL DEFAULT 2.0,
        max_depth INTEGER DEFAULT 50,
        samples_per_pixel INTEGER DEFAULT 100,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (user_id) REFERENCES users (user_id)
            ON DELETE CASCADE
    );
    """

    with self.connect() as conn:
        conn.execute(create_users_table)
        conn.execute(create_user_preferences_table)
        conn.execute(create_render_preferences_table)
```

Our SQL queries use parameterized statements (? placeholders), which helps prevent SQL injection attacks and improves query efficiency. The method catches `sqlite3.IntegrityError`, which could occur if e.g. a username already exists. This prevents my app from crashing, which allows us to instead provide useful feedback for the user to change their username choice.

Default values for each field, across all tables, ensure that every user has a complete set of preferences from the start - without any initial user input being required when their account is created (apart from their username and password). We return the new `user_ID` if the operation is successful and `None` if there is an error. This allows the code which calls the method to easily check if the operation was successful - which prevents it trying to access the new `user_ID` and data without it existing.

The SQL statements are defined and executed separately which improves their readability/maintainability. Only the necessary data (`username` and `password_hash`) is passed using parameters, following the principle of least privilege.

All of these things well suit the method for a relational database and provide clear feedback on the success or failure of the operation.

The `update_security` method uses optional parameters to provide flexibility without needing separate methods for each update scenario.

```python
def add_user(self, username: str, password_hash: str) -> Optional[int]:
    """Add a new user and create their default preferences."""
    sql_user = "INSERT INTO users (username, password_hash) VALUES (?, ?)"
    sql_user_prefs = "INSERT INTO user_preferences (user_id) VALUES (?)"
    sql_render_prefs = "INSERT INTO render_preferences (user_id) VALUES (?)"

    try:
        with self.connect() as conn:
            cursor = conn.execute(sql_user, (username, password_hash))
            user_id = cursor.lastrowid
            conn.execute(sql_user_prefs, (user_id,))
            conn.execute(sql_render_prefs, (user_id,))
            return user_id
    except sqlite3.IntegrityError:
        return None
```

The SQL statement is dynamic, based on the method parameters, so only necessary fields are updated and updates are checked if they are actually needed before proceeding - avoiding unnecessary database calls.

No sensitive info is exposed in the return value, only success or failure. This method focuses solely on updating security settings, adhering to the Single Responsibility Principle.

```python
def update_security(
  self,
  user_id: int,
  username: Optional[str] = None,
  password_hash: Optional[str] = None
) -> bool:
    """Update security settings (username and/or password)."""
    if not username and not password_hash:
        return False

    update_fields = []
    values = []
    if username:
        update_fields.append("username = ?")
        values.append(username)
    if password_hash:
        update_fields.append("password_hash = ?")
        values.append(password_hash)

    sql = f"""
    UPDATE users
    SET {', '.join(update_fields)},
        updated_at = CURRENT_TIMESTAMP
    WHERE user_id = ?
    """
    values.append(user_id)

    try:
        with self.connect() as conn:
            cursor = conn.execute(sql, values)
            return cursor.rowcount > 0
    except sqlite3.IntegrityError:
        return False
```

Both the `update_user_preferences` and the `update_render_preferences` methods follow similar design principles and justifications - with different fields being edited.

For the `get_all_settings` method, `LEFT JOIN`s are used to ensure data is returned even if preference entries don't exist - preserving the user record while keeping related preferences optional. Using explicit column selection rather than using `SELECT *` for better performance and clarity.

I could consider returning nested dictionaries for each preference type. This would represent the relationship between data better, however, this is not a necessary addition at this time.

The `get_all_user` method also uses the same design philosophy as the method above.

```python
def get_all_settings(self, username: str) -> Optional[Dict]:
    """Retrieve all settings for a user."""
    sql = """
    SELECT
        u.user_id, u.username, u.created_at,
        up.theme, up.auto_save,
        rp.render_name, rp.image_width, rp.aspect_ratio,
        rp.focus_distance, rp.aperture, rp.max_depth,
        rp.samples_per_pixel
    FROM users u
    LEFT JOIN user_preferences up ON u.user_id = up.user_id
    LEFT JOIN render_preferences rp ON u.user_id = rp.user_id
    WHERE u.username = ?
    """
    with self.connect() as conn:
        cursor = conn.execute(sql, (username,))
        row = cursor.fetchone()
        return dict(row) if row else None
```

Database Singleton

```python
class DatabaseSingleton:
    """
    A singleton class that manages database connections.
    Ensures only one instance of the database connection is created and reused.
    """
    _instance: Optional['DatabaseSingleton'] = None
    _db_manager = None

    def __new__(cls):
        """
        Controls instance creation to enforce the singleton pattern.

        Returns:
            DatabaseSingleton: The single instance of this class.
        """
        if cls._instance is None:
            cls._instance = super(DatabaseSingleton, cls).__new__(cls)
        return cls._instance

    def __init__(self):
        """
        Initializes the database manager if it doesn't exist yet.
        Creates necessary directories and establishes the database connection.
        """
        if self._db_manager is None:
            project_root = Path(__file__).parent.parent
            db_path = project_root / "data" / "3Dev.db"
            db_path.parent.mkdir(parents=True, exist_ok=True)
            self._db_manager = DataManager(str(db_path))

    @property
    def db(self):
        """
        Provides access to the database manager.

        Returns:
            DataManager: The database manager instance.
        """
        return self._db_manager
```

The `DatabaseSingleton` class ensures that only one instance of the database is created and used throughout the source code. This is done by setting up the database, `.db`, file using the `data_manager` code and a python class to control it from.

The use of the `Path` object for constructing the database path is platform-independent, which reduces the likelihood of causing errors by changing where this file is in the project directory. Making use of encapsulation, the `_db_manager` is private with a public accessor through the `db`, `@property`.

The `get_db` function provides an interface for other areas of the application to use to make use of the database without having to understand or use the singleton mechanics correctly - which reduces the risk of errors throughout my code base.

I have decided to implement classes for handling the key operations needed throughout my code which need to be linked to the database. These “middle men” interfaces mean that database management code and higher level application interface code can change without affecting one another.

```python
def get_db():
    """
    A helper function to access the database manager.

    Returns:
        DataManager: The database manager instance from the DatabaseSingleton.
    """
    return DatabaseSingleton().db
```

The code in these classes is kept very minimal to fulfill this robustness idea - all of the logic for handling data is kept in the `data_manger` so that these interface classes only have to pass through variables to be changed or checked by the database (and return back the resulting appropriate values).

```python
class UserManager:
    """
    Provides an interface for and manages user-related creation and retrieval operations.
    """

    def __init__(self):
        """
        Initializes the UserManager with a database connection.
        """
        self.db = get_db()

    def create_user(self, username: str, password_hash: str) -> Optional[int]:
        """
        Creates a new user in the database.

        Parameters:
            username (str): The username for the new user.
            password_hash (str): The hashed password for the new user.

        Returns:
            Optional[int]: The ID of the newly created user, or None if creation failed.
        """
        return self.db.add_user(username, password_hash)

    def get_user(self, username: str) -> Optional[Dict]:
        """
        Retrieves security information for a user by username.

        Parameters:
            username (str): The username of the user to retrieve.

        Returns:
            Optional[Dict]: A dictionary containing user security information,
                            or None if the user doesn't exist.
        """
        return self.db.get_user_section(username, 'security')
```

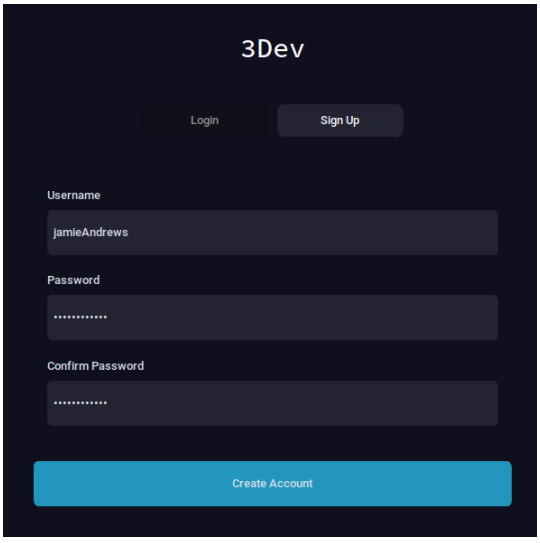
The structure of the `SettingsManager` and `RenderManager` is very much the same as the `UserManager` so I will not include redundant explanations.

Testing

Completed Tests

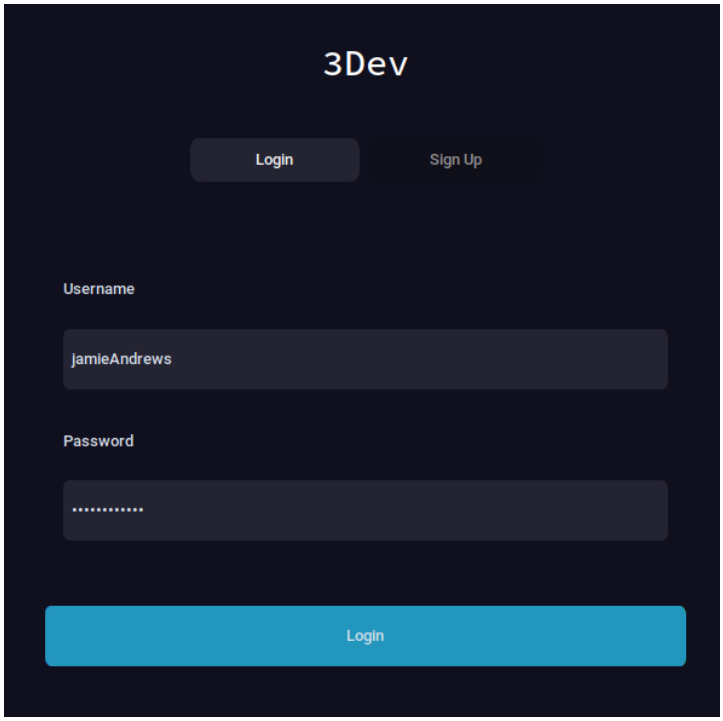
1.1	Successful sign up takes user to dashboard - ensures page redirection working correctly	Normal	Submit sign up with valid credentials	User redirected to dashboard and username displayed in top right	Multiple Users

Valid username and password seen entered into the given fields

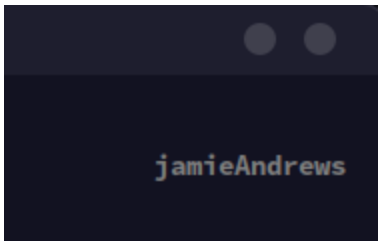


Dashboard correctly opened to the version control menu by default, with the username in the top right (showing that the sign up correctly processed the user data)

1.21	Successful login takes user to dashboard - ensures sign-up adds user to database and user data is fetched correctly	Normal	Submit login with valid credentials for user created in test 1.1	User redirected to dashboard and username displayed in top right	Multiple Users



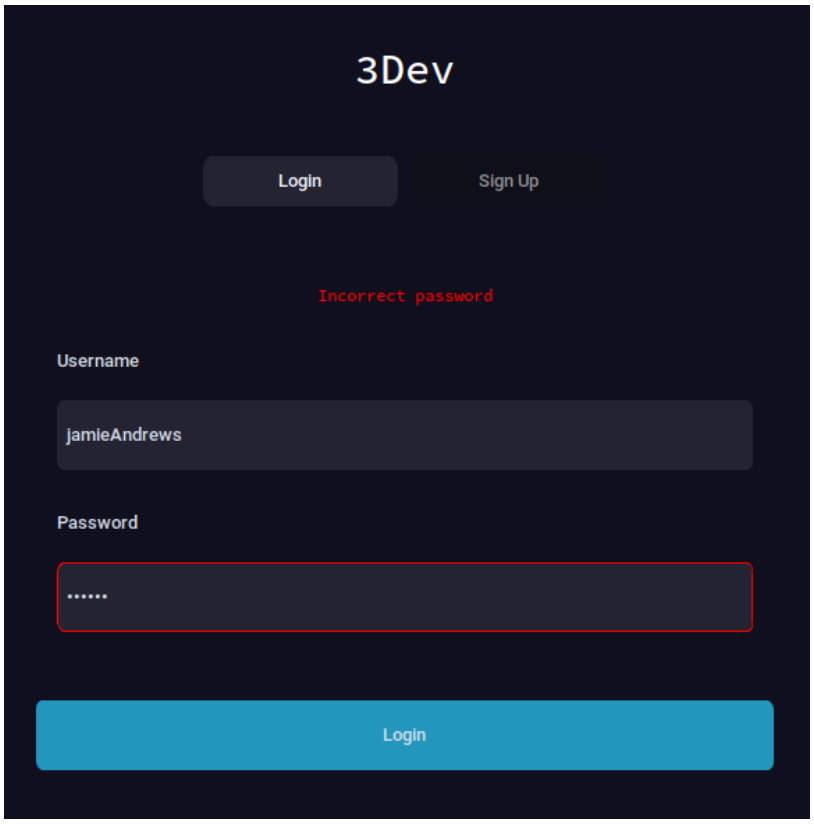
Correct username and password inputted to login section using the newly created account details



Dashboard once again opened, with the username displayed to show successful login to the correct account

1.22	Failed login does not take user to dashboard - ensures sign-up adds user to database and user data is fetched correctly (which also ensures security of accounts and their data)	Erroneous	Submit login with incorrect password for user created in test 1.1	User not redirected to dashboard and error message displayed	Multiple Users

Previously created account username inputted along with incorrect password (clearly evidenced by the fewer characters) and appropriate error message displayed without user being redirected to the dashboard

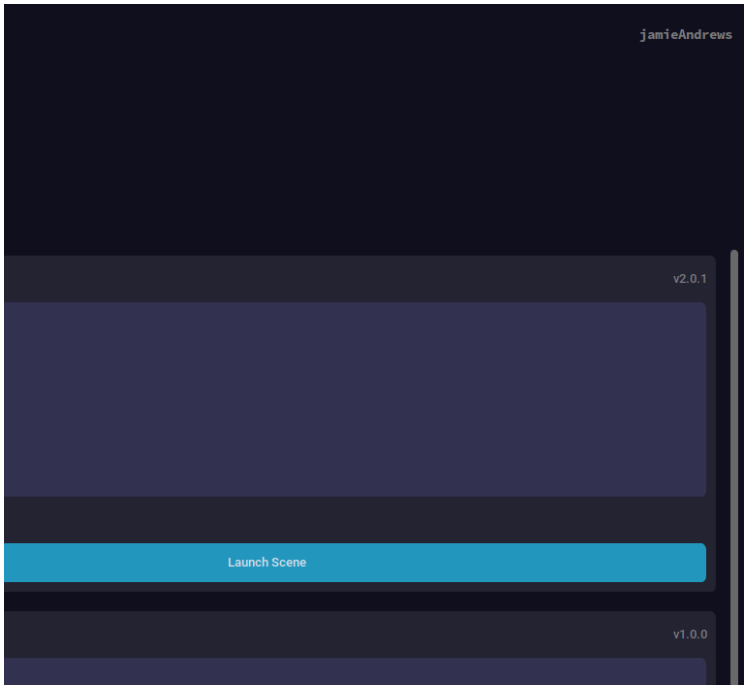


3.1	Altering settings changes database records - ensures settings are remembered and saved between logins	Normal	User edits one of the settings fields	Database field updated	Settings Menu

After altering preferences in the settings menu, the correct values are displayed after the following login.

The saving of these changes can also be seen via the command line output as they happen.

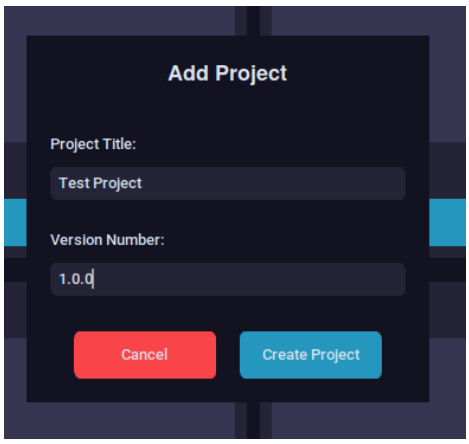
4.1	Version control menu scrollable - in order to view all projects on one menu page	Normal	User drags scroll bar down	Project cards grid scrolls down	Version Control



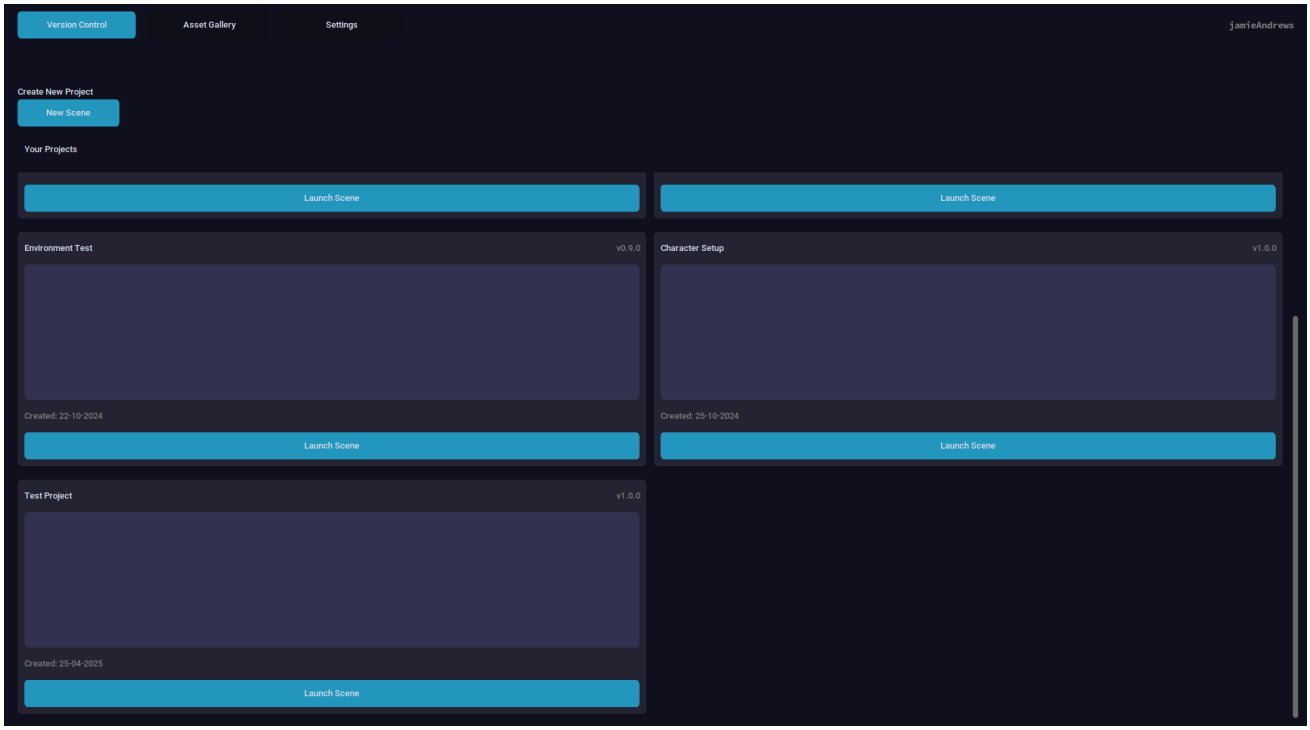
Scroll bar located on the right hand side of the window, only covering the project cards section of the menu.

4.21	“New Scene” button properly displays object addition dialogue - ensures button click event triggers the proper menu	Normal	“New Scene” button clicked	Project addition menu displayed	Version Control

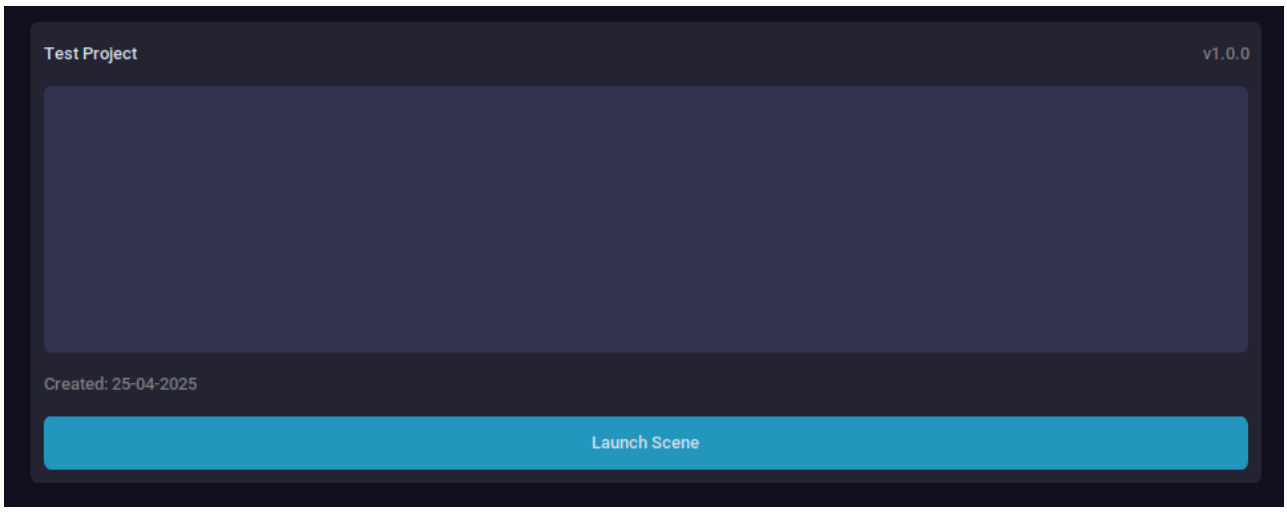
“New Scene” button correctly opens project addition dialogue with fields to fill in project information.



4.22	Project cards update with database (addition) - ensure version control menu is accurate and up to date	Normal	Project info filled and “Create Project” button clicked	Project card correctly shown in version control menu	Version Control

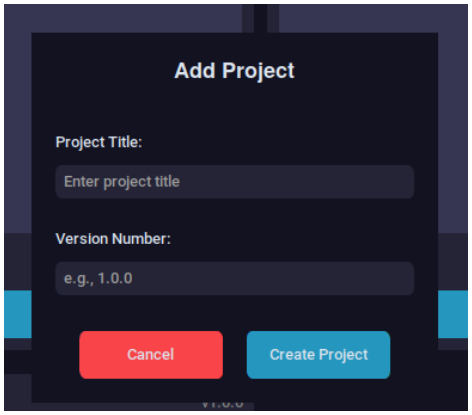


New project card added to gallery after new project submission (additional evidence of scrollable frame now that there are enough projects being displayed to scroll through)

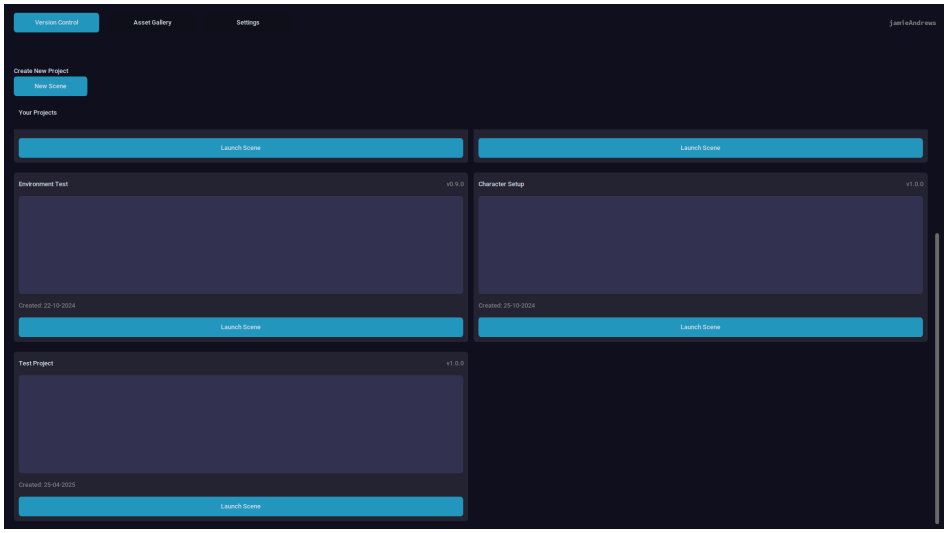


Correct project information, provided via the project addition dialogue shown above, stored and displayed. Additionally, the date of creation is accurately determined and displayed without the need for user input.

4.23	Project addition menu does not add projects without info being added - ensures unnamed and unidentifiable (empty) projects being added to database and menu	No Input	Project info not filled and “Create Project” button clicked	Project cards remain unchanged (no new card added)	Version Control

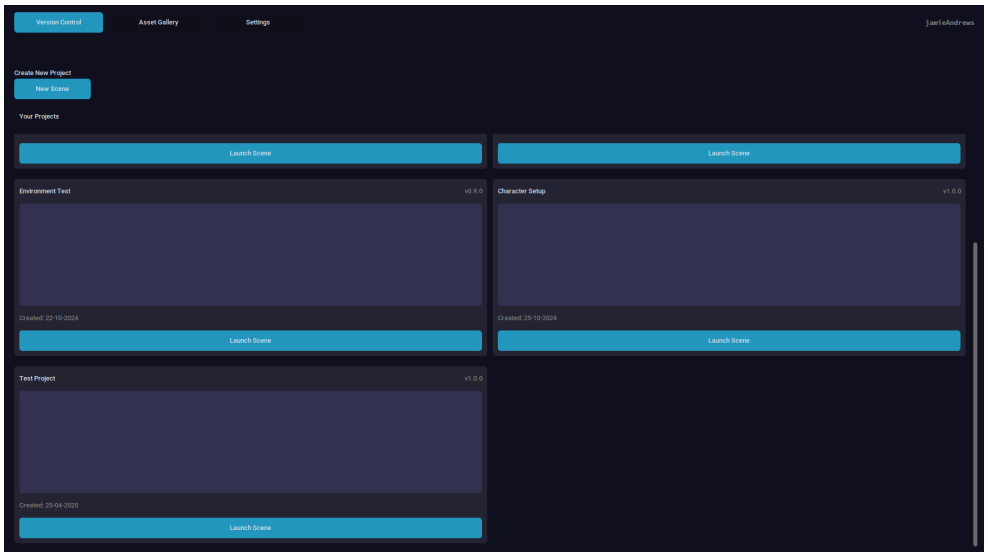


Project fields left blank.

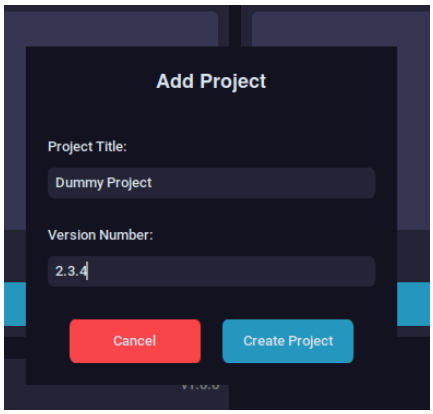


Project gallery remains as it was without an empty card being added.

4.24	Project addition menu “Cancel” button closes menu without any change to project gallery (despite the presence of project info in input fields) - ensures proper interaction without project addition dialogue (verifying that the Cancel button closes the dialogue) and simultaneously ensures no projects get unintentionally added to the gallery	Normal	Project info filled and “Cancel” button clicked	Project addition dialogue closes and cards remain unchanged (no new card added)	Version Control



Project gallery remains unchanged and new project dialogue no longer displays in the centre of the screen.



Project info fields filled, followed by the clicking of the cancel button.

Failed Tests

The following tests were failed due to a lack of time to implement all of the intended requirements, possible future remedies will be covered during the evaluation section.

2.1	Asset cards presented alphabetically in the grid - makes it easier for user to find their assets without searching	Normal	Assets added to the database and user navigates to asset gallery	Asset cards listed alphabetically by asset name	Imports

2.21	Asset searching filters asset card display results	Normal	User types asset name into search bar (case insensitive)	Only asset with that name shown in gallery	Imports

2.21	Lack of asset sharing results in no filters - ensures users can see all their assets	No Input	User types nothing into search bar	All assets shown in gallery	Imports

2.31	Asset cards update with database (addition) - ensure asset gallery is accurate and up to date	Normal	Asset added to database	Asset cards shown in version control menu	Imports

2.32	Asset cards update with	Normal	Asset removed from database	Asset cards shown in	Imports

	database (subtraction) - ensure asset gallery is accurate and up to date			version control menu	

3.2	Logging out of account - lets users swap between accounts without having to restart the application	Normal	User clicks logout button	Redirected to sign-in page	Settings Menu

4.23	Project cards update with database (subtraction) - ensure version control menu is accurate and up to date	Normal	Project removed from database	Project no longer shown in version control menu	Version Control

Post-Sprint Analysis

I performed mini evaluations for each of the previous sprints during the “Sprint Analysis” sections of their respective following sprints. As there is not another sprint after sprint 3, I will cover a short analysis/conclusion on sprint 3.

After the development and testing of sprint 3 I have connected all the user features, current version control features and some settings menu features to the database. This means the asset gallery is only the front end UI.

Evaluation

This section is to evaluate my 3D scene rendering software (3Dev) against the initial requirements and success criteria. This will consider test outcomes, usability, maintenance, and potential future developments.

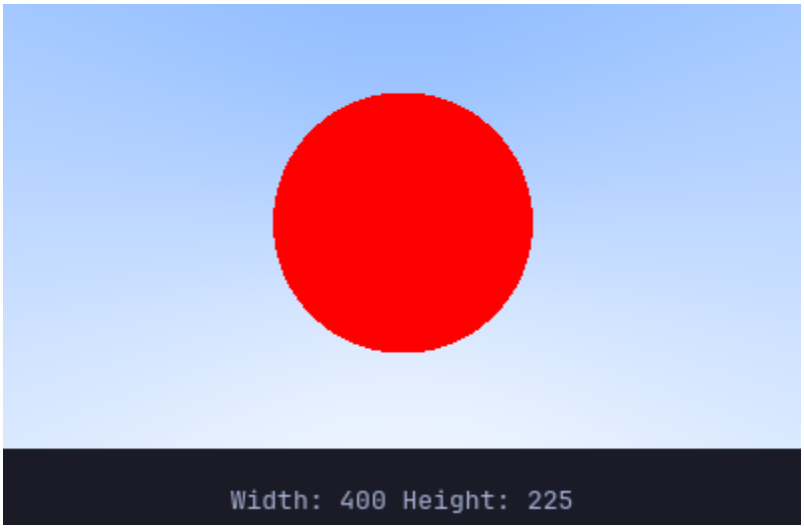
Post-Development Testing

Post-development testing was conducted throughout the project using an agile methodology, with specific tests planned and executed at the end of each sprint (Sprint 1, Sprint 2, Sprint 3). Evidence for functional, robustness, and usability testing is documented within the 'Testing' sections of each sprint report.

Functional Testing:

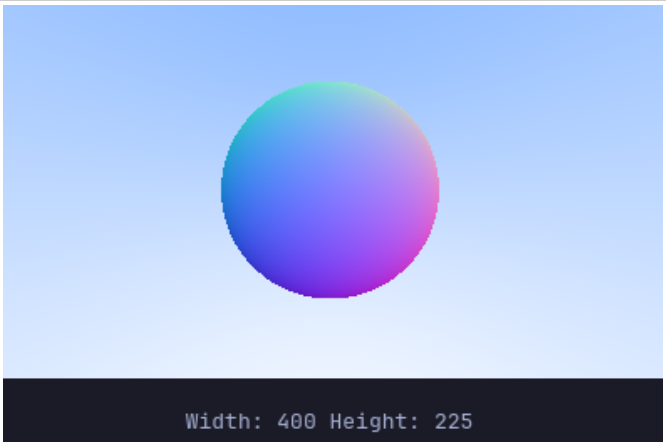
- Functional tests focused on verifying core features like sphere/quadrilateral intersection, material rendering (diffuse, metal, dielectric), camera positioning, defocus blur, and anti-aliasing. Robustness was implicitly tested through error handling in user input (e.g.,

login/signup validation) and file handling (e.g., import validation planned but not fully implemented).



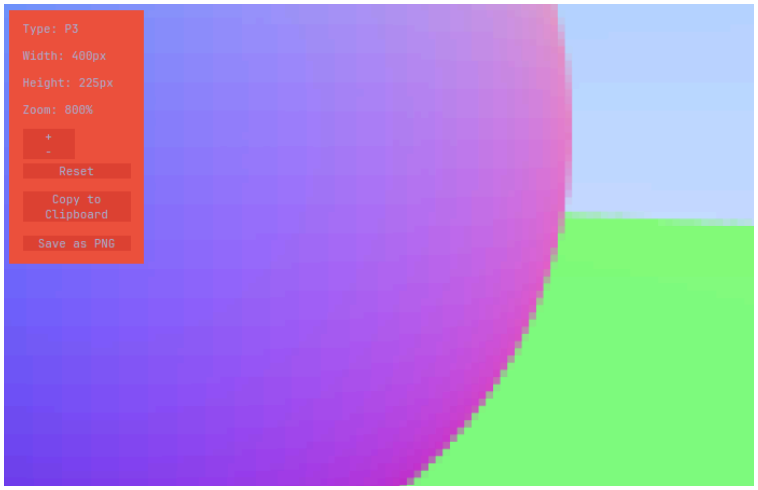
Functional Test:

3.1	Rendering sphere - makes sure sphere intersections are being calculated correctly	Normal	Add a sphere object to the scene, click render button	Rendered image of a sphere produced	Sphere Intersections



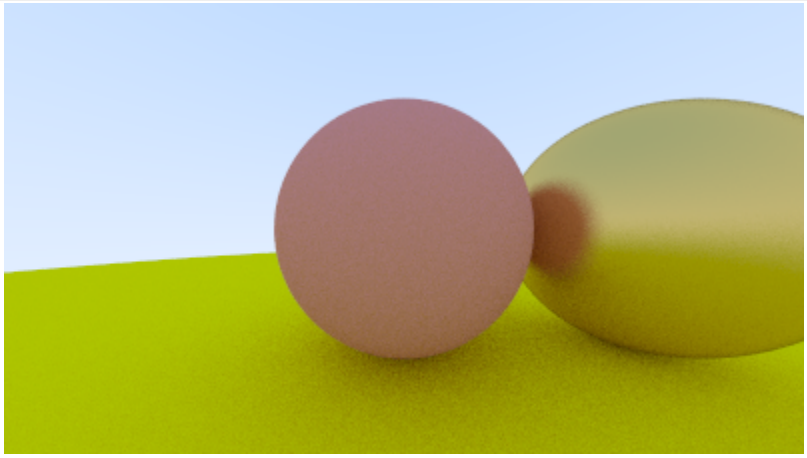
Completed Test:

4	Surface normal gradient - Displays the mapping between surface normal values and points on the sphere	Normal	Add a sphere object to the scene with surface normal texture, click render button	Rendered image of sphere with smooth colour gradient	Surface Normals



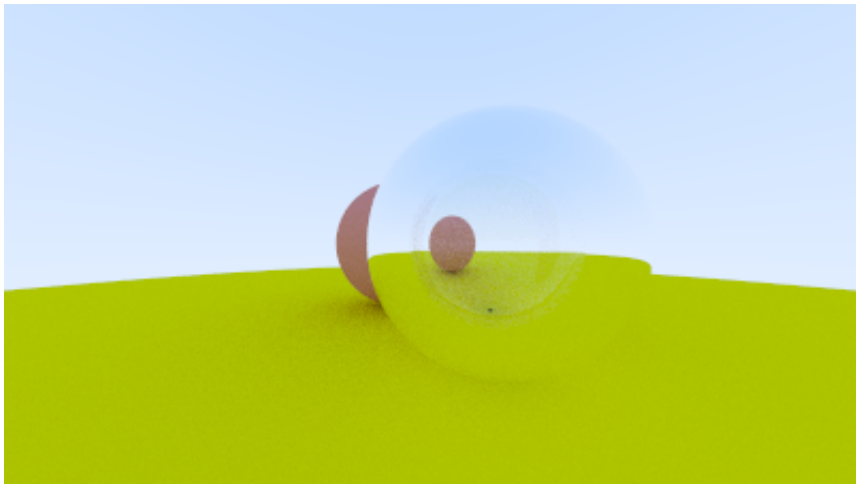
Completed Test:

5	Edges with antialiasing - ensures multiple samples are taken for each pixel	Normal	Add sphere to scene and position camera close to the edge, click render	Rendered image of sphere with smooth edge between the object and background	Antialiasing



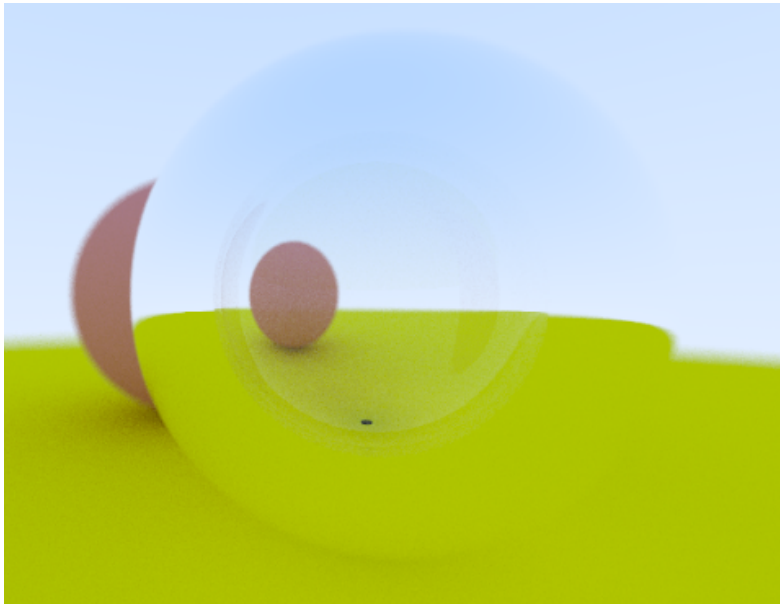
Completed Test:

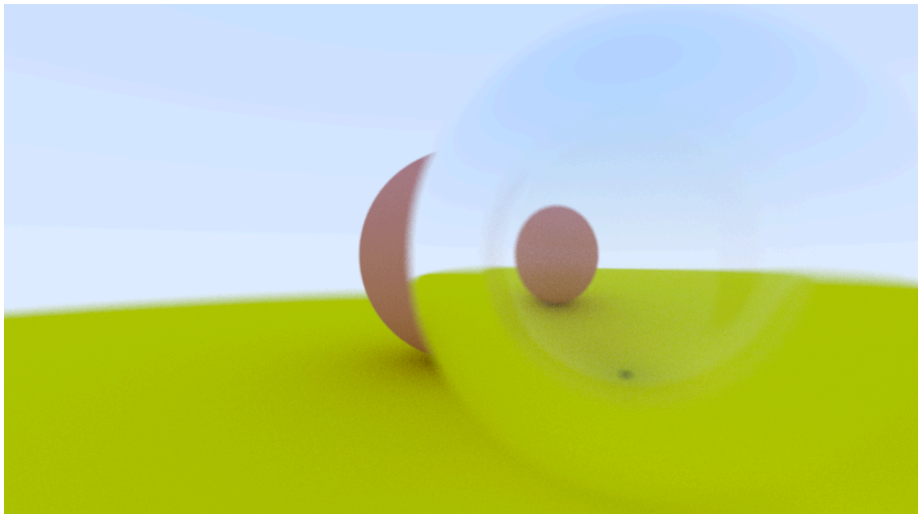
7	Metal sphere reflection - shows how rays are being specularly scattered off smooth surfaces	Normal	Add sphere with metal material next to a sphere with matte material, click render	Rendered image of matte sphere in the reflection of the metal sphere	Reflections



Completed Test:

8	Hollow glass sphere - Demonstrates the successful reflection and refraction of rays incident on dielectrics	Normal	Add glass sphere in front of matte sphere to scene, click render button	Rendered image of distorted matte sphere seen through the glass sphere	Dielectrics



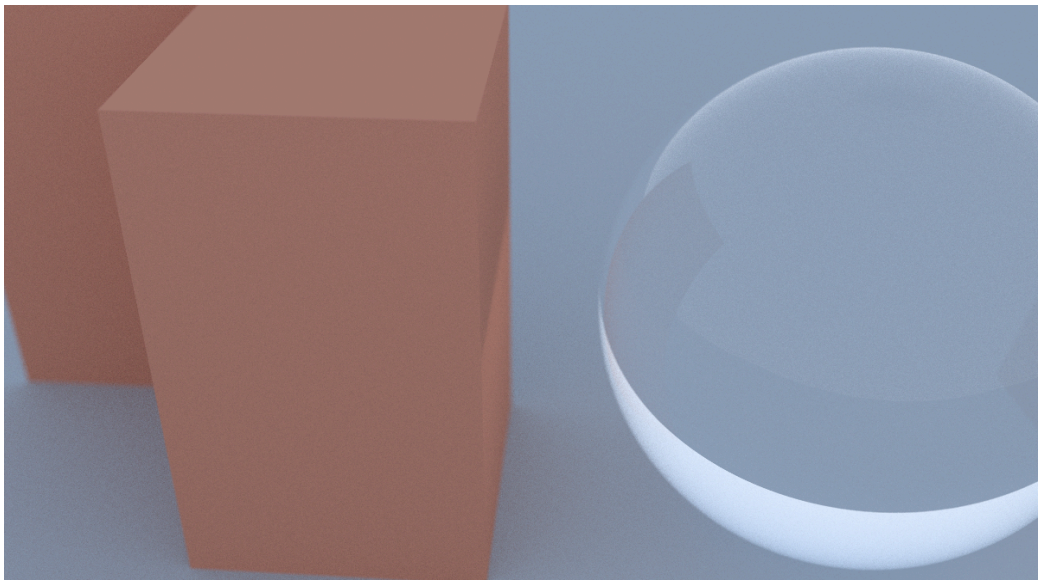


Completed Tests:

12.11	Changing depth of field - FOV - Checks FOV is being calculated and applied correctly	Normal	Render 2 identical scenes with different FOVs	Scene with larger FOV should have a larger range of depths in focus	Defocus Blur
12.12	Changing depth of field - shifting - Checks the right band of depths is being focused	Normal	Render 2 identical scenes with the DOF in different places	One scene should focus on foreground while the other focuses on the background	Defocus Blur

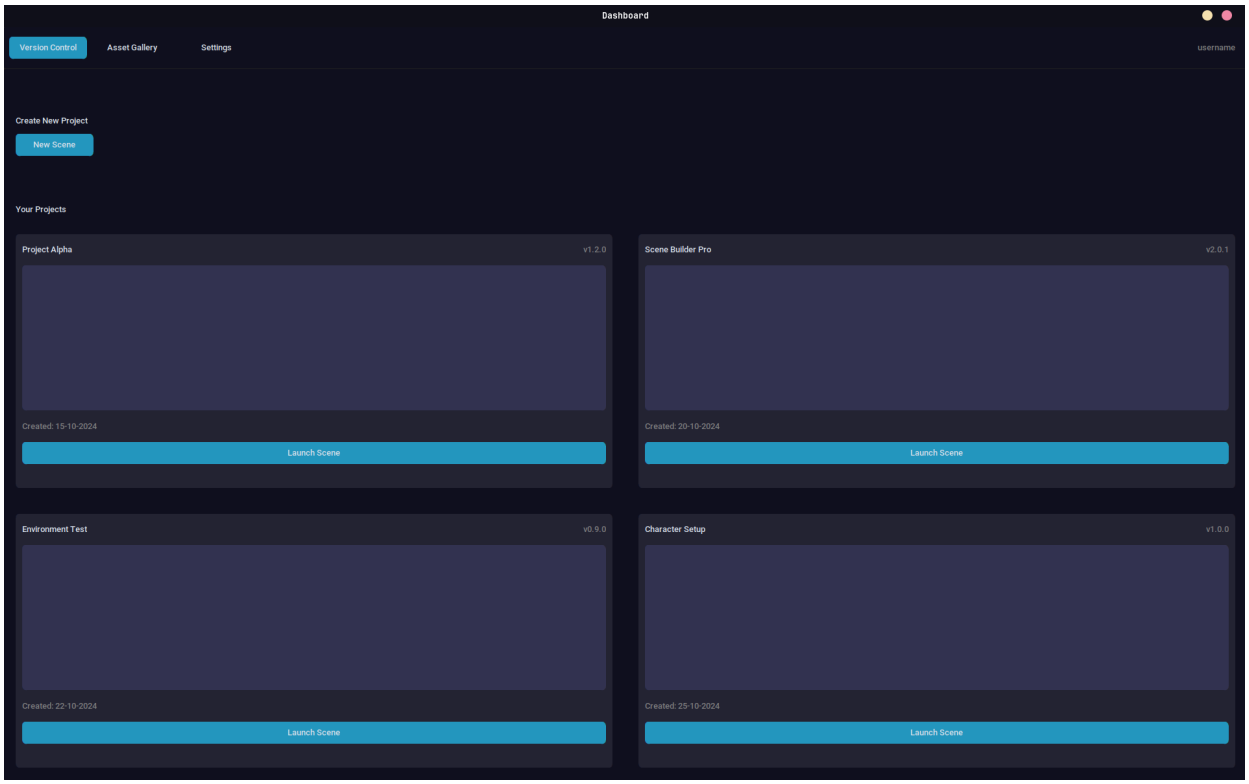
Completed Test:

2.11	New viewing position - changes camera pos when new coords given	Normal	Edit camera viewing position (valid coordinates)	Scene renders from new position	Positionable Camera



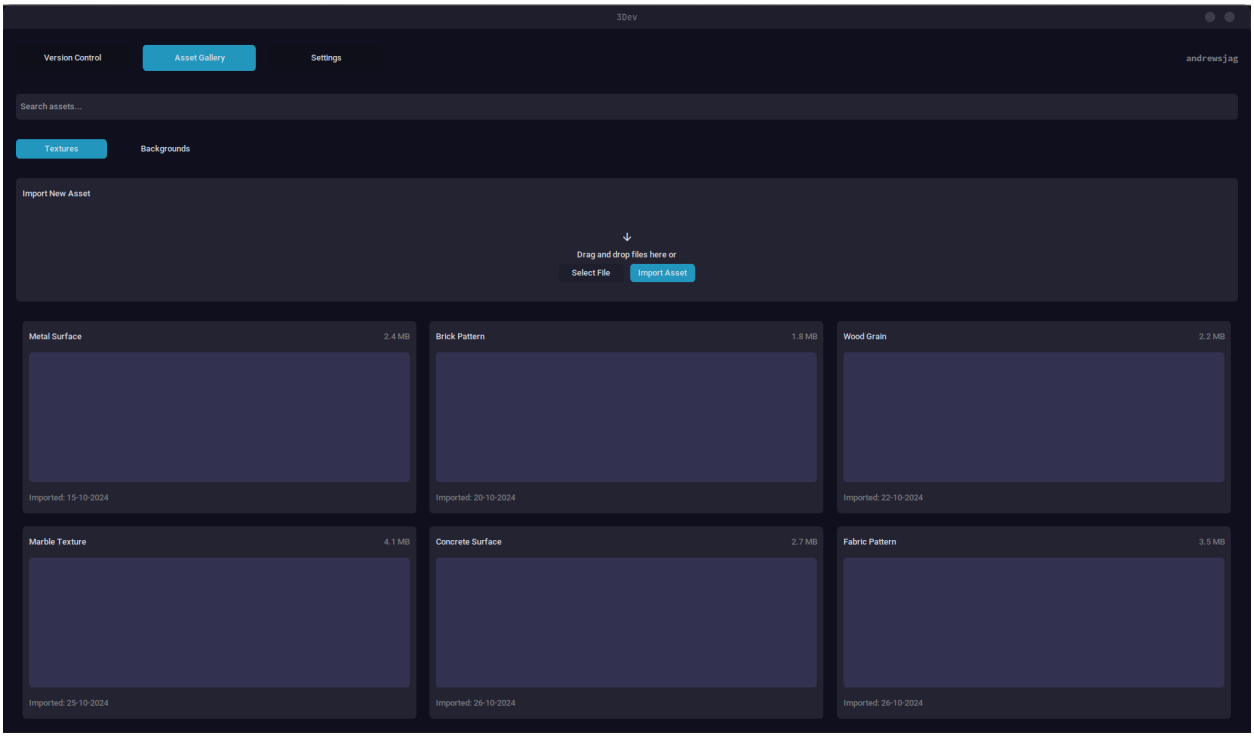
This output completes the 3D cuboid test. Normal data inputted (cuboid centre and dimensions given to cuboid helper function) - relating to the "Quadrilaterals" requirement.

The expected output of a matte 3D cuboid displayed has been achieved.



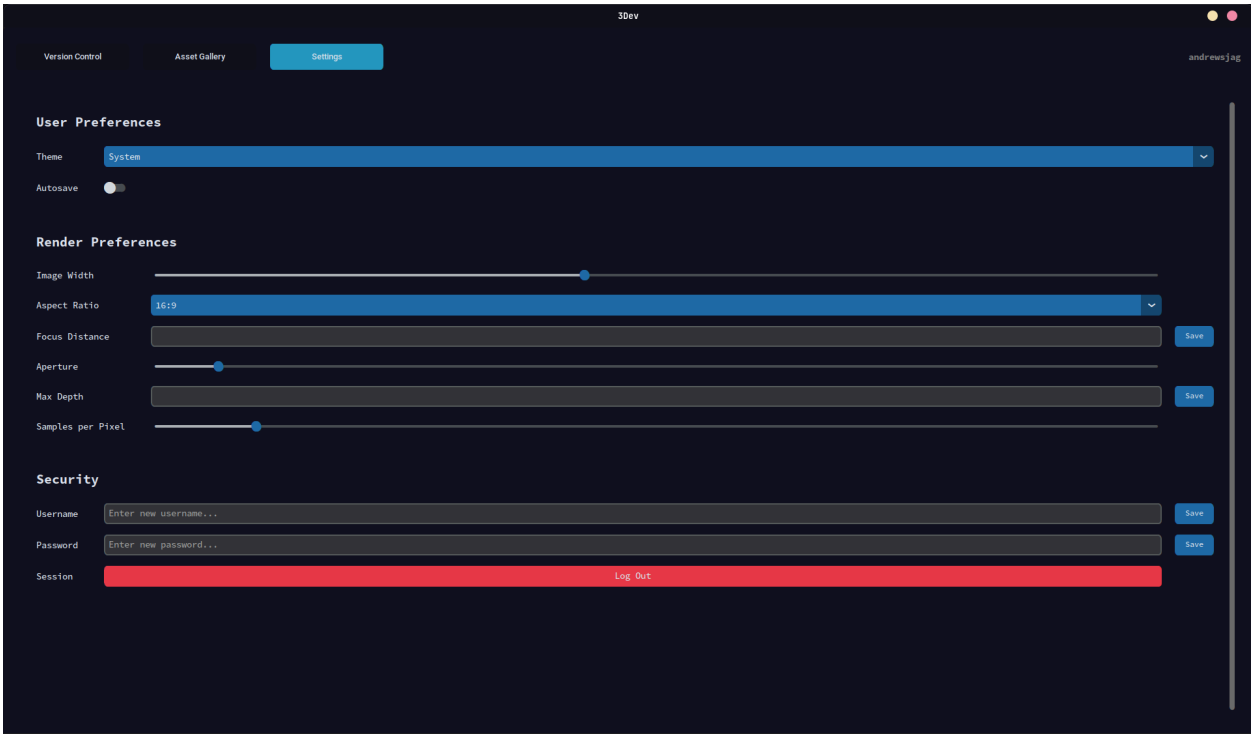
Completed Test:

17.3	Navigation to version control - allows users to access the dashboard form the settings menu and gallery	Normal	user clicks on version control button	User redirected to version control	Dashboard



Completed Test:

17.1	Navigation to gallery - allows users to access the asset gallery from the main dashboard and settings menu	Normal	user clicks on gallery button	User redirected to asset gallery	Dashboard



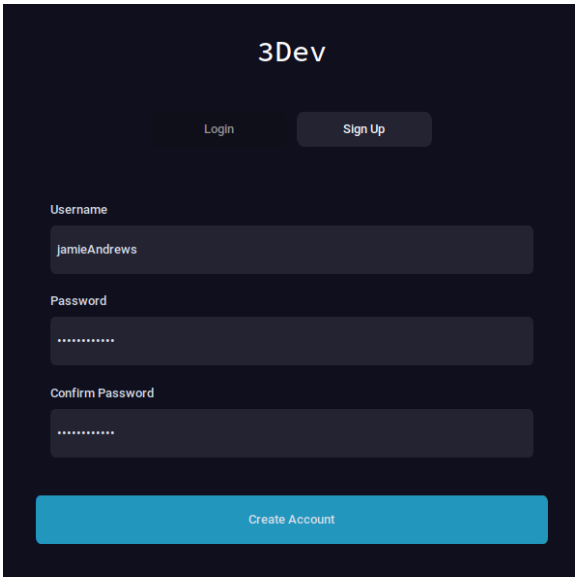
Completed Tests:

17.2	Navigation to settings menu - allows users to navigate to the settings menu from the main dashboard and gallery	Normal	user clicks on settings button	User redirected to settings page	Dashboard

(As can be seen on the right hand side of the page, the window is scrollable, however there are not enough settings present for this to have any effect).

17.4	Scrollable Settings - allows user to view all settings easily	Normal	user drags scroll bar	Page scrolls up/down	Dashboard

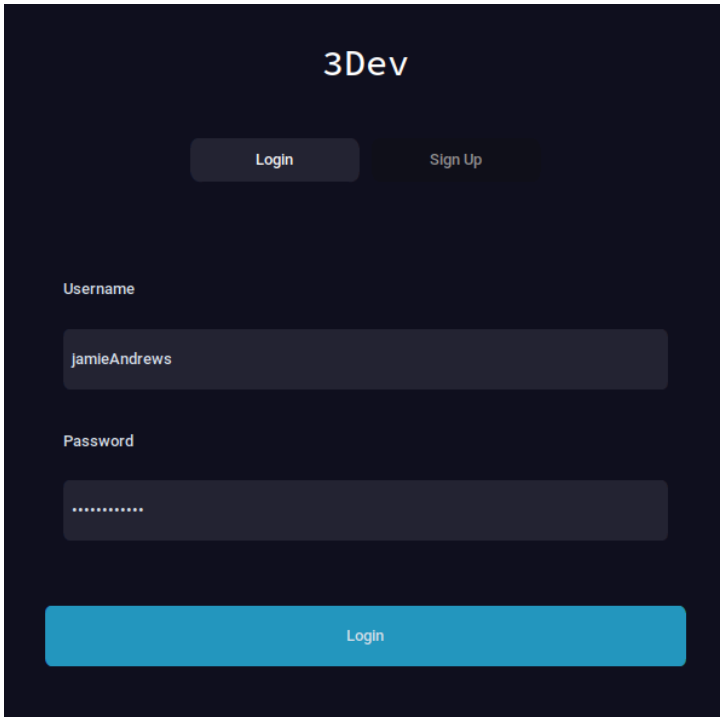
Valid username and password seen entered into the given fields



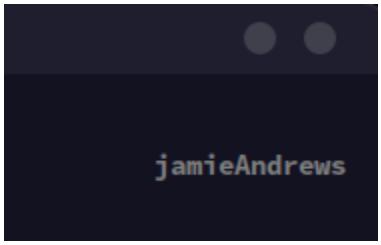
Dashboard correctly opened to the version control menu by default, with the username in the top right (showing that the sign up correctly processed the user data)

Completed Test:

1.1	Successful sign up takes user to dashboard - ensures page redirection working correctly	Normal	Submit sign up with valid credentials	User redirected to dashboard and username displayed in top right	Multiple Users



Correct username and password inputted to login section using the newly created account details



Dashboard once again opened, with the username displayed to show successful login to the correct account

Completed Test:

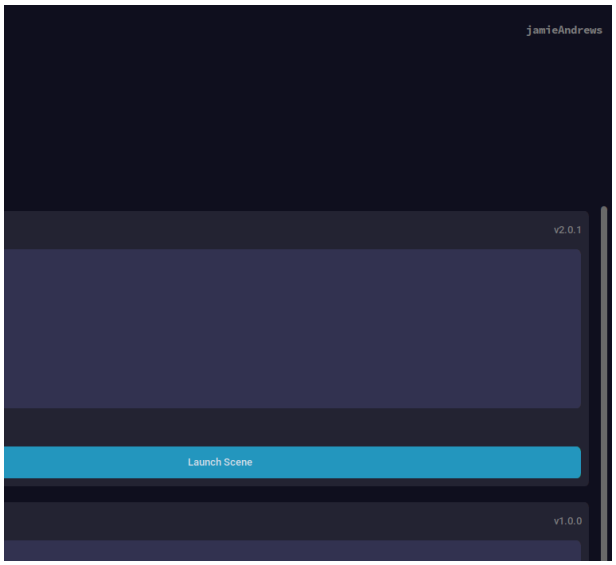
1.21	Successful login takes user to dashboard - ensures sign-up adds user to database and user data is fetched correctly	Normal	Submit login with valid credentials for user created in test 1.1	User redirected to dashboard and username displayed in top right	Multiple Users

After altering preferences in the settings menu, the correct values are displayed after the following login

The saving of these changes can also be seen via the command line output as they happen.

Completed Test:

3.1	Altering settings changes database records - ensures settings are remembered and saved between logins	Normal	User edits one of the settings fields	Database field updated	Settings Menu

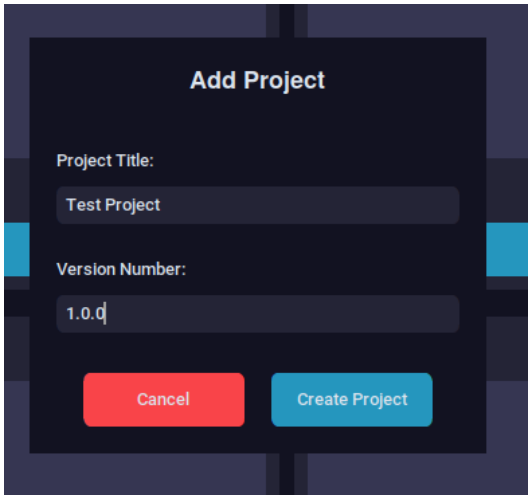


Scroll bar located on the right hand side of the window, only covering the project cards section of the menu.

Completed Test:

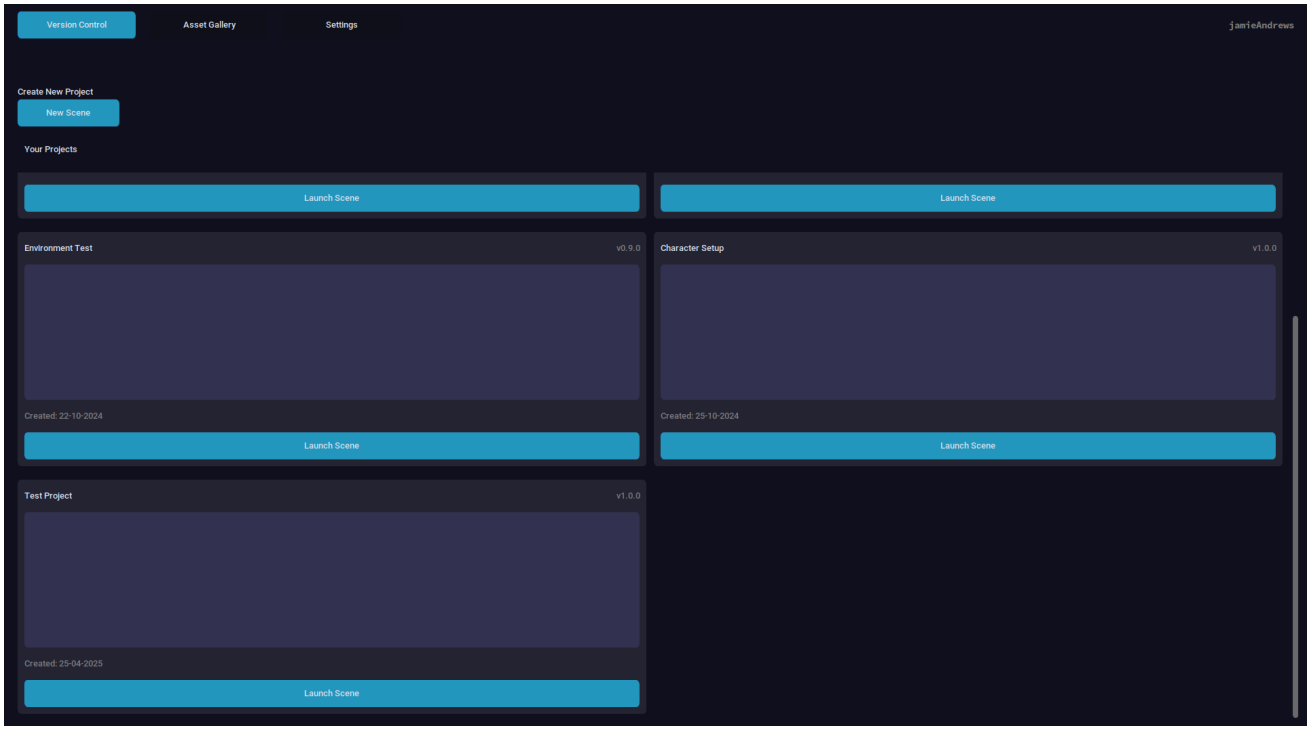
4.1	Version control menu scrollable - in order to view all projects on one menu page	Normal	User drags scroll bar down	Project cards grid scrolls down	Version Control

“New Scene” button correctly opens project addition dialogue with fields to fill in project information.



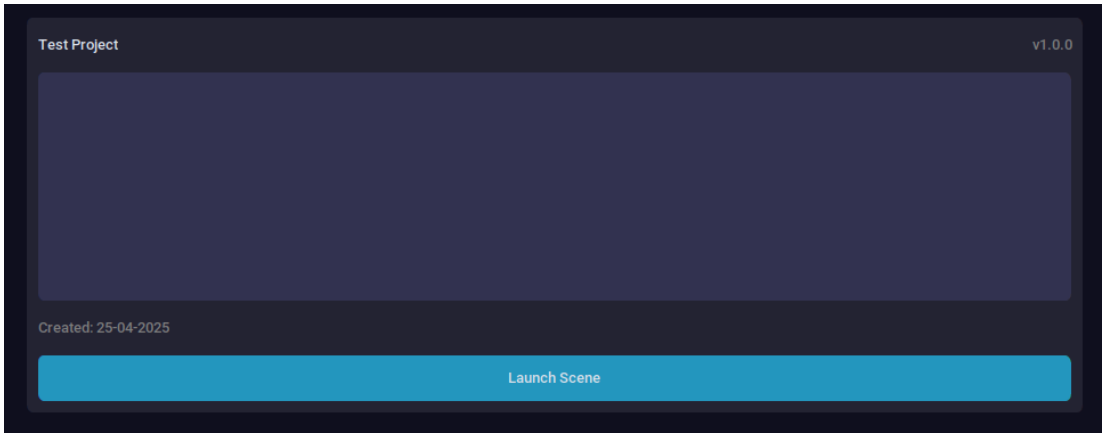
Completed Test:

4.21	“New Scene” button properly displays object addition dialogue - ensures button click event triggers the proper menu	Normal	“New Scene” button clicked	Project addition menu displayed	Version Control



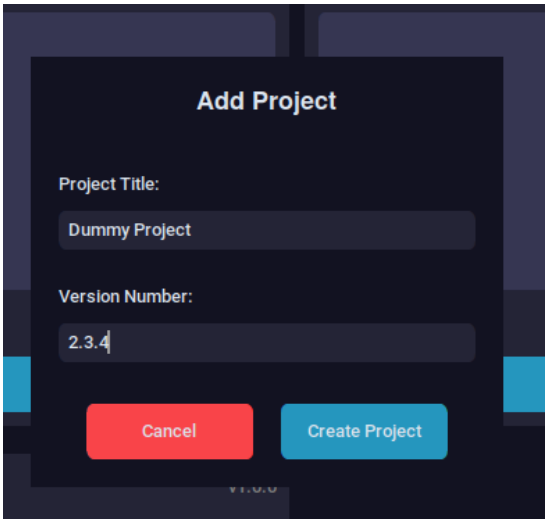
New project card added to gallery after new project submission (additional evidence of scrollable frame now that there are enough projects being displayed to scroll through)

Correct project information, provided via the project addition dialogue shown above, stored and displayed. Additionally, the date of creation is accurately determined and displayed without the need for user input.



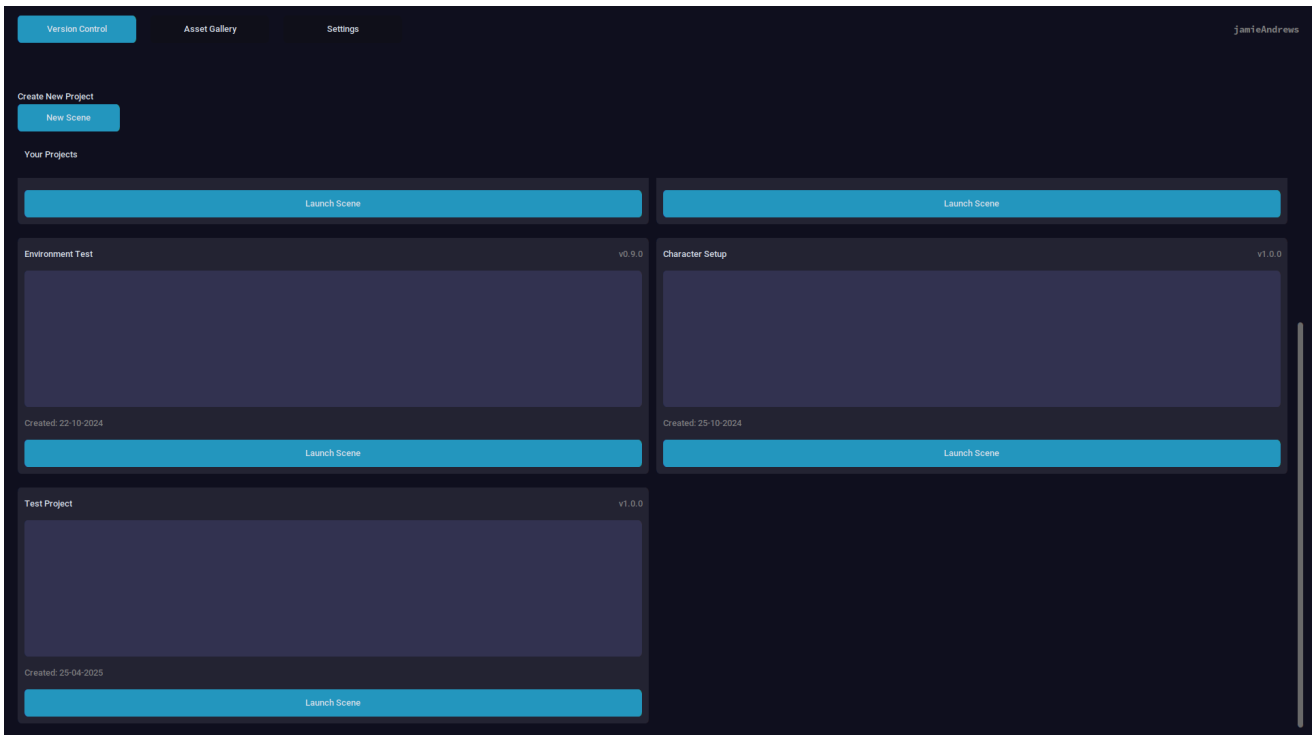
Completed Test:

4.22	Project cards update with database (addition) - ensure version control menu is accurate and up to date	Normal	Project info filled and “Create Project” button clicked	Project card correctly shown in version control menu	Version Control



Project info fields filled, followed by the clicking of the cancel button.

Project gallery remains unchanged and new project dialogue no longer displays in the centre of the screen.



Completed Test:

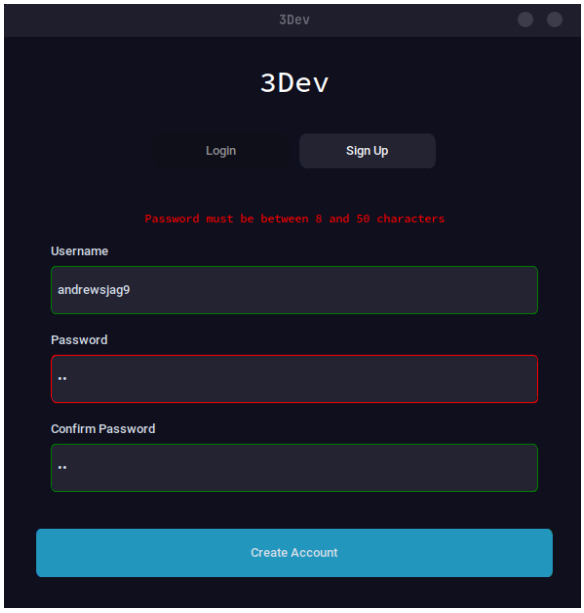
4.24	Project addition menu “Cancel” button closes menu without any change to project gallery (despite the presence of project info in input fields) - ensures proper interaction without project addition dialogue (verifying that the Cancel button closes the dialogue) and simultaneously ensures no projects get unintentionally added to the gallery	Normal	Project info filled and “Cancel” button clicked	Project addition dialogue closes and cards remain unchanged (no new card added)	Version Control

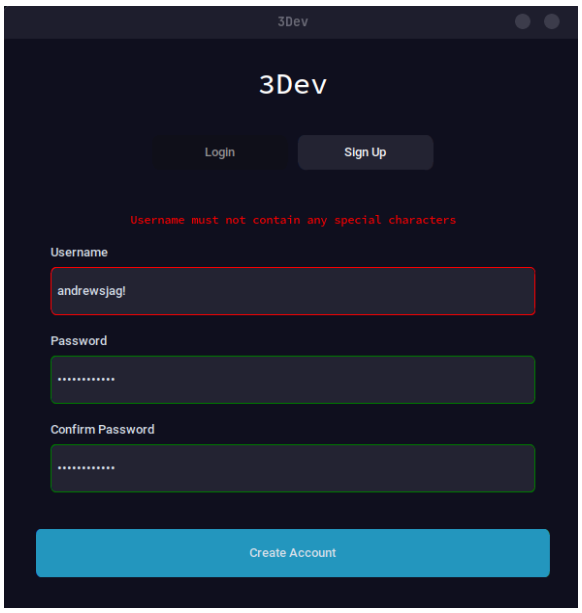
Robustness Testing:

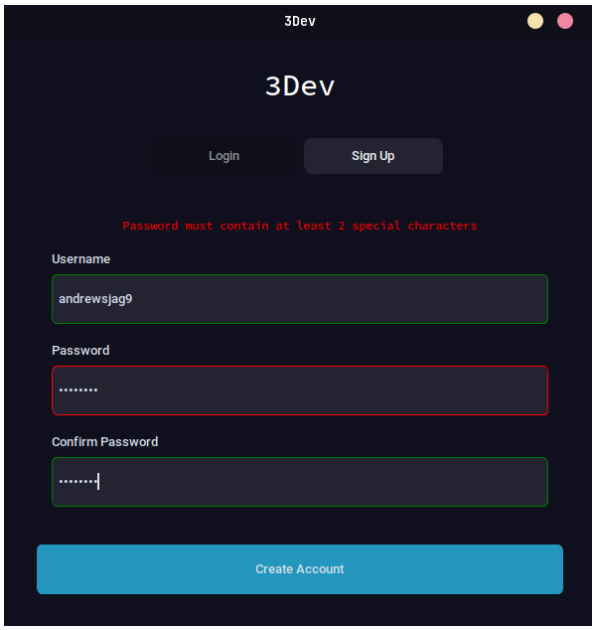
9.14	Login fail - empty fields - stops users logging in with empty fields	No Input	username: empty password: empty	Error message: "Please fill login fields"	Multiple Users
9.21	Signup fail - invalid pass length - increases security	Erroneous	username: validUsername password: 123	Error message: "password must be at least 6 characters"	Multiple Users
9.22	Signup fail - invalid username (special characters) - increases security	Erroneous	username: invalidUsername password: validPassword Conf: =password	Error message: "Username cannot contain special characters"	Multiple Users
9.23	Signup fail - invalid pass (not enough special characters) - increases security	Erroneous	username: validUsername password: 987hfaii Conf: =password	Error message: "password must be at least 2 special characters"	Multiple Users
9.24	Signup fail - invalid pass (>3 consecutive digits) - increases security	Erroneous	username: validUsername password: 1234mnb!! Conf: =password	Error message: "password must have no more than 3 consecutive digits"	Multiple Users
9.25	Signup fail - passwords don't match - increases security and avoids user error	Erroneous	username: validUsername password: validPass Conf: != password	Error message: "passwords do not match"	Multiple Users
9.26	Signup fail - unfilled fields - stops users making accounts with blank properties	No Input	username: empty password: empty Conf: empty	Error message: "passwords do not match"	Multiple Users

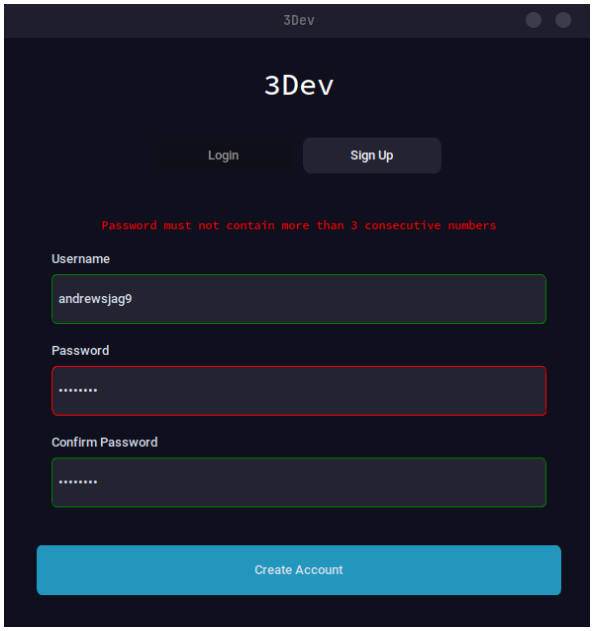
9.21, 9.22, 9.23, 9.24, 9.25, 92.6 (in order):

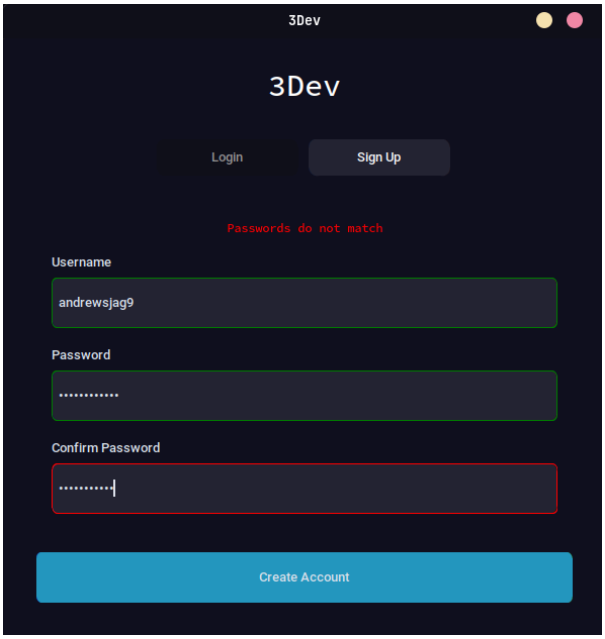
Sign up validation

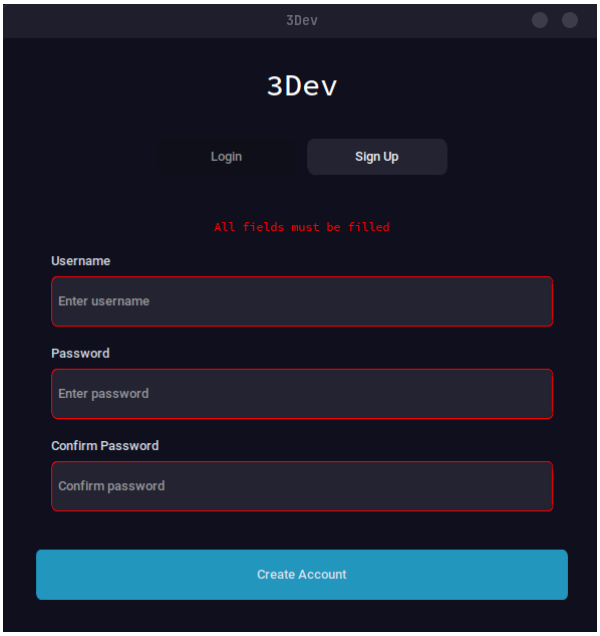

3Dev
3Dev
Login
Sign Up
Password must be between 8 and 50 characters
Username
andrewsjag9
Password
Confirm Password
Create Account


3Dev
3Dev
Login
Sign Up
Username must not contain any special characters
Username
andrewsjag!
Password
Confirm Password
Create Account


3Dev
3Dev
Login
Sign Up
Password must contain at least 2 special characters
Username
andrewsjag9
Password
Confirm Password
Create Account

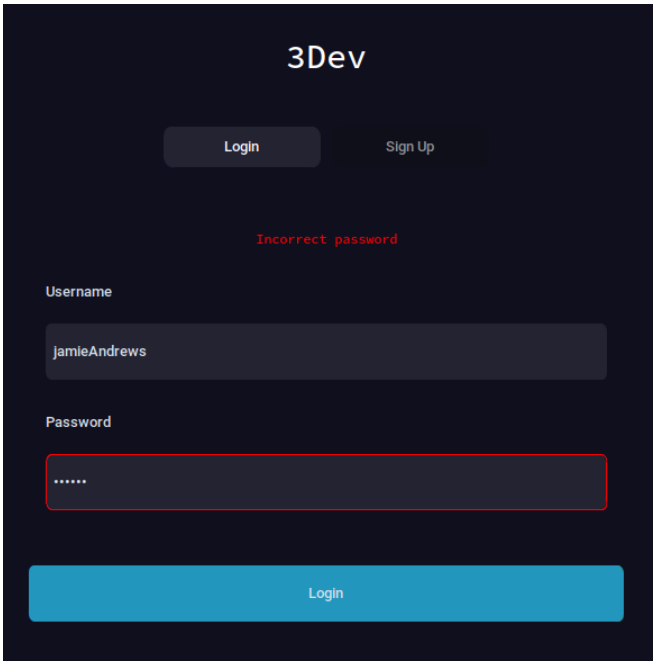

3Dev
3Dev
Login
Sign Up
Password must not contain more than 3 consecutive numbers
Username
andrewsjag9
Password
Confirm Password
Create Account





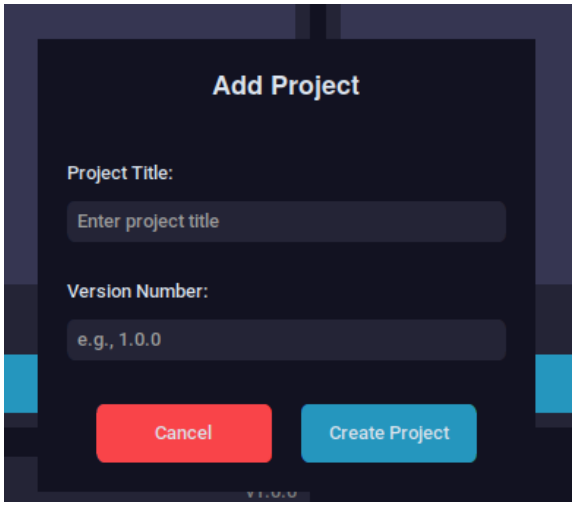
(End of sign up tests)

previously created account username inputted along with incorrect password (clearly evidenced by the fewer characters) and appropriate error message displayed without user being redirected to the dashboard



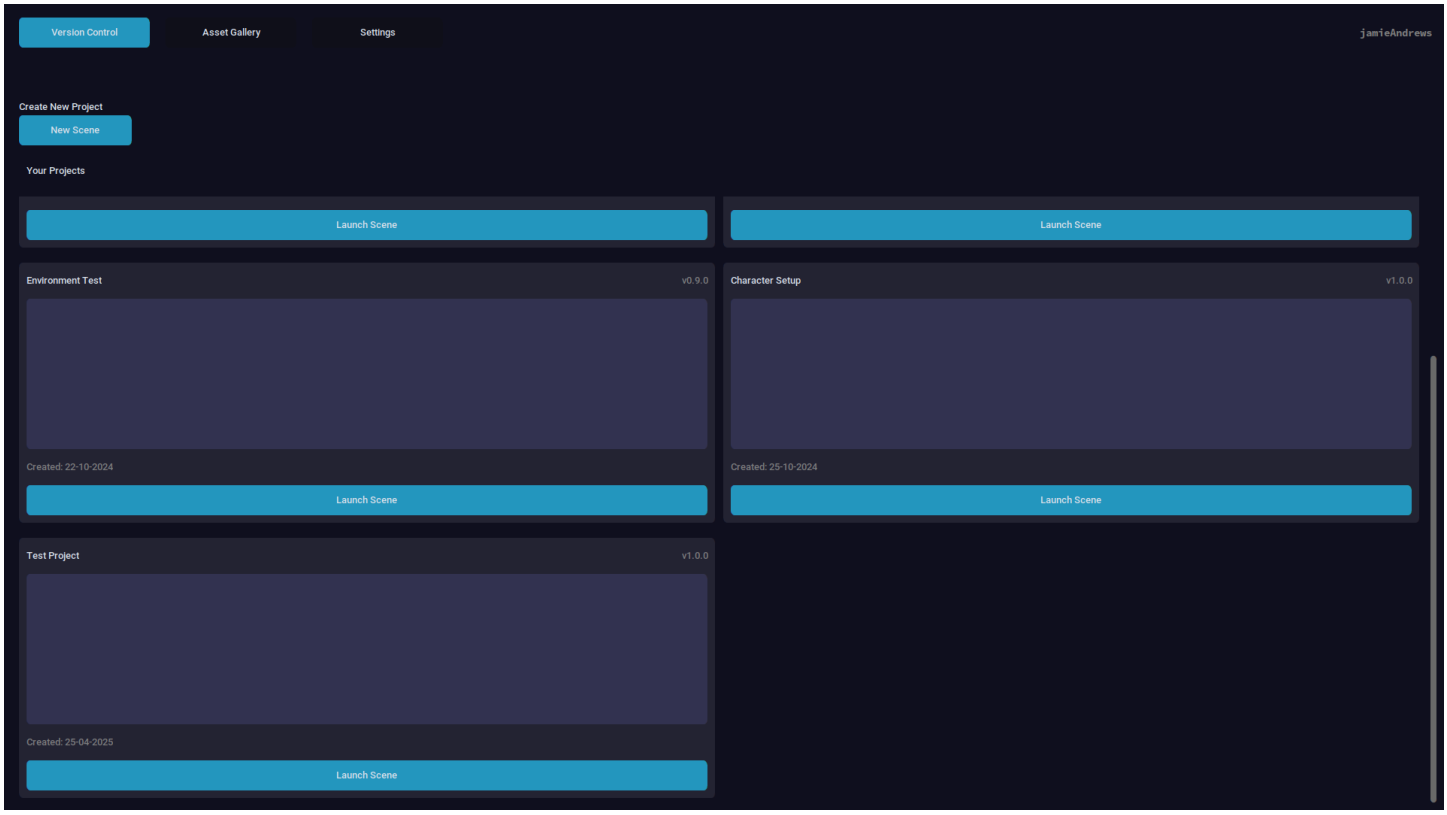
Completed Test:

1.22	Failed login does not take user to dashboard - ensures sign-up adds user to database and user data is fetched correctly (which also ensures security of accounts and their data)	Erroneous	Submit login with incorrect password for user created in test 1.1	User not redirected to dashboard and error message displayed	Multiple Users



Project fields left blank.

Project gallery remains as it was without an empty card being added.



Completed Test:

4.23	Project addition menu does not add projects without info being added - ensures unnamed and unidentifiable (empty) projects being added to database and menu	No Input	Project info not filled and “Create Project” button clicked	Project cards remain unchanged (no new card added)	Version Control

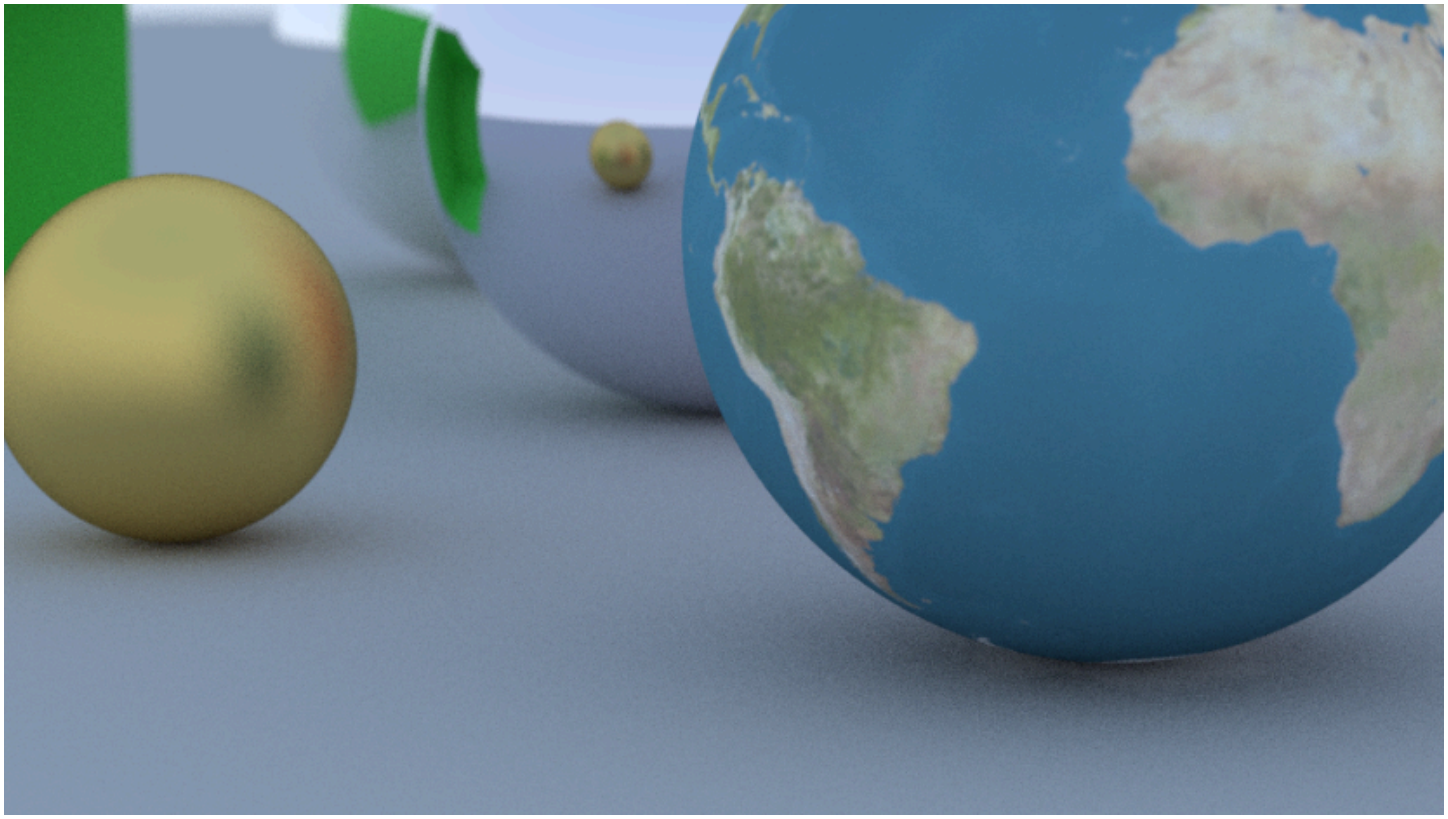
Usability Testing:

Usability testing was primarily conducted through stakeholder feedback during sprint analysis and specific tests on UI navigation and interaction. Stakeholder Jason Wang provided feedback on the UI design, leading to iterative improvements in the sign-in page layout and aesthetics. Each stakeholder also gave feedback at the end of their sprint-to-sprint testing which guided what I prioritised in the following sprint.

Now that development has finished, I have given Jason and Martin their respective areas of the software to test and assess in order to gain overall usability feedback.

Martin

Despite the ray tracing portion of the software not being linked to the UI due to time constraints, because of Martin's programming background, he was still able to test the ray tracer by editing values of objects and camera properties in the `main` function of `main.rs`. He put together a scene in order to try and test out all of the different shape and material features covered by the ray tracer.



This was the scene Martin created, after which I gave him a questionnaire to get his feedback on each feature he implemented.

Path Tracing:
Q1: "What do you think of the quality of the path tracing?"

A1: “All of your fundamental concepts are well done so i’m getting clean images that look physically accurate, the only issue is that rendering takes some time so it's not that practical to be using a high resolution or depth (which would get you the really crisp scenes)”

Q2: “What are the most important things you think could be changed or added?”
A2: “I think for future changes the goal is to add more effects like fluids and lens artifacts but the first thing you need to do is focus on optimisation because the run time is already relatively slow so adding anything more before optimisation will make it almost unusable”

Textures:
Q1: “What did you think of the quality of textures in your scene?”
A1: “Textures were all good to be honest, only thing which would have improved them is general resolution increased I already mentioned and maybe using a higher res texture image for the earth map”

Q2: [Martin already mentioned changes in his A1 response]

Effects:
Q1: “How did you find the quality of the effects which were covered?”
A1: “The defocus blur does a lot to make the image look much more photorealistic, and especially helps if you're making a scene with a particular key focus because you can slightly blur the less important objects surrounding it.”

Q2: “How would you like to see the effects changed or improved?”
A2: “Obviously there are some missing effects, most notably ambient occlusion (which I think should be the next priority), but I think the more important change would be to figure out a way to alter focus distance in real time. I found that it was tricky to get the focus distance right because I'd have to use trial and error by changing the value with lengthy render times.”

Jason

[TestVideo.mp4]

Jason was given the python UI application to try and test, once he had understood all of the features he produced a short video demonstrating the UI features which have been covered.

This includes catching invalid sign-up attempts [0:18, 0:24] → successful sign-up [0:28] → dashboard navigation [0:32 - 0:36] → invalid login attempt caught [0:49] → successful login [0:53] → successful project creation [1:07] → successful project creation + tiling [1:17] → editing settings menu options [1:26 - 1:46] → demonstrating the successful saving of changed preferences to the database [1:52 - 1:56]

After his time testing the UI, I performed a similar questionnaire/interview to the one with Martin:

Sign-in
Q1: “What was the general sign-in experience like?”
A1: “I think the sign-in is excellent, not much I can say that is wrong with it, all runs smoothly”

Q2: “Is there anything you would change about it?”
A2: “The only downside to having a rigorous credential validation system is that it might take a few tries when signing up to come up with a valid password so it might be good if there was a way to see what requirements you need to meet without just doing trial and error.”

Dashboard
Q1: “What was the experience with all the dashboard menus?”
A1: “The version control menu is all smooth, asset gallery looks nice but obviously isn't functional right now and the settings menu is a bit of both, works well but is missing a bit of functionality in some places”

Q2: “What could be better about the dashboard menus?”
A2: “Other than adding functionality to the asset gallery and the missing pieces of the settings menu, I think the only thing I’d change is the fact that the version control menu is entirely blank until you add a project so it might be useful to add some filler text or something until it gets filled in.”

Success Criteria Evaluation

The detailed success criteria outlined at the start of each sprint were used as a basis for testing, I will use the feedback I've just received from Jason and Martin to assess how well each of these have been achieved.

- **High-Level Success Criteria (from Analysis section):**
    1. **Intuitive Settings & Main Menu:** Partially Met. The dashboard navigation (Version Control, Assets, Settings) is functional (Tests 17.1-17.3) and the sign-in process is clear. However, the scene creator interface was not developed, and the settings menu functionality is only partially implemented (database saving works, but logout and full functionality is missing).

    2. **Gallery Asset Handling (>=100):** Not Met. The Asset Gallery UI was created, but the underlying functionality (database integration, searching, handling >100 assets, importing/exporting) was not completed in time (Failed Tests 2.1-2.32).

    3. **Render Quality & Speed:** Partially Met. The ray tracer produces high-quality images with various materials and effects which were discussed in earlier designs (Tests 3.1-8, 12.11-12.12). BVH was implemented for optimization (Sprint 1 - Programming - BVH). However, there were issues with render times as the BVH system was not fully developed and was only in a more basic form. This lack of render speed is a high priority for changes that could be made in the future.

    4. **Multi-User Support:** Mostly Met. The login/signup system functions correctly, allowing different users to access the dashboard (Tests 1.1, 1.21). User data (settings, projects) is correctly stored in the database. However, features which cross over to version control like merging scenes with other users have not been implemented.

5. **Visually Appealing 3D Environment:** Not Met. The 3D “Scene Creator” environment was not developed due to time constraints shifting focus to core UI features as well as database integration in Sprints 2 and 3.

6. **Version Control Management:** Partially Met. The UI for displaying projects exists, and new projects can be added via a dialogue, storing basic info in the database (Tests 4.21, 4.22). Although, core versioning features (saving versions, comments, branching, merging, chronological display) were not implemented due to a lack of time, once again.

7. **Handling Many Objects:** Not Met. The scene creator UI for adding/editing objects was not developed. The renderer's BVH tries to support lots of objects but this hasn’t been fully tested since it already struggles (time wise) with a more limited number of objects in the scene.

8. **Accurate Material/Texture Mapping:** Partially Met. The renderer supports various materials (matte, metal, glass) and basic texture mapping (e.g. earth map). However, importing and applying custom image textures via the UI was not completed as it was part of the asset gallery (which only got covered on the front end).

9. **Multiple Render Scheduling:** Not Met. This feature (Render Queue/Parallel Rendering) was identified but not implemented in the available time.

10. **Swift Asset Import/Export:** Not Met. The Asset Gallery UI was designed, but import/export functionality was not implemented as previously mentioned (Failed Tests 2.1-2.32).

Addressing Partially/Unmet Criteria:

- **Gallery/Asset Handling:** Complete database integration for assets, implement file import/validation logic, and develop search/filtering functionality. This is essential for allowing users to take advantage of the texture mapping features and allowing for the production of fully fledged scenes.

- **Render Speed:** Implement the connection between the UI and the Rust renderer so that the app can be used as one whole unit. Conduct benchmark tests comparing render times with/without different BVH methods like the ones covered by Sabastian in the research section. These results can then be compared to competitor software to see how well the renderer has been optimised.

- **3D Environment/Object Handling:** Develop the Scene Creator UI using PyThreeJS, allowing for object addition, manipulation, and editing. All of these actions need to be linked to the

renderer and database.

- **Version Control:** Implement database logic for saving distinct project versions, storing metadata (such as comments and timestamps), as well as the mentioned branching/merging functionality. The version control menu would also need adding to in order to display information relating to these versions

- **Render Scheduling:** Develop UI elements and backend logic to manage a queue of render tasks. The exact method for achieving this was not entirely figured out as it was only a potential feature rather that was lower priority compared to the ones above it.

Maintenance Issues

- **Code Modularity:** The Python UI code makes use of classes and utility modules (e.g., DataManager, DatabaseSingleton, UI utilities). These aid maintainability by separating parts of the code base which do not directly rely on each other. The Rust renderer code is also structured so that it can very easily be made into separate modules, however it is still currently all in one file. Modularity will make it easier to update or debug individual parts without affecting the whole code base. Additionally, separating the code into multiple files makes it easier for other developers to understand how my code base is split up and how each section interacts with the others.

- **Dependencies:** The project relies on external libraries (CustomTkinter, PIL, Glam, etc.). Updates to these libraries could introduce changes which require code adjustments. Using a package manager like uv helps manage Python dependencies and stay on top of updates. This is less of a major concern as many libraries are developed to be backwards compatible so anything that works in an older version will still work in the newer versions.

- **Database Schema:** Changes to requirements might necessitate alterations to the SQLite database schema. As the database grows it may be necessary to redesign the database structure in order to ensure it complies with 3NF. Keeping to this standard avoids data getting too complicated to manage as the amount of data being stored grows.

- **Error Handling:** Error handling is present (i.e. SQL integrity errors, input validation), but more comprehensive error logging/reporting could improve how maintainable the project is for diagnosing issues as the UI (and database) expands.

Limitations

- **Still Images Only:** The software does not support video rendering or animation. This is a feature that many high quality rendering software provides now so it may be beneficial to look at implementing this in order to keep up with competitors.

- **No Real-Time Rendering:** Once the Scene Creator is implemented, ray tracing only occurs when the user renders the scene using the “Render” button. If ray tracing were able to be computed and displayed in real time, that may speed up the workflow of users creating scenes. Although, this would likely require internet connection to external cloud computing to avoid people being limited by their own hardware capabilities.

- **No Cross-Device Interaction:** User accounts and projects are currently stored locally; no cloud synchronization is used which means real-time collaboration cannot occur. This is not ideal for groups of people working on larger projects that need to be worked on in parallel by multiple people.

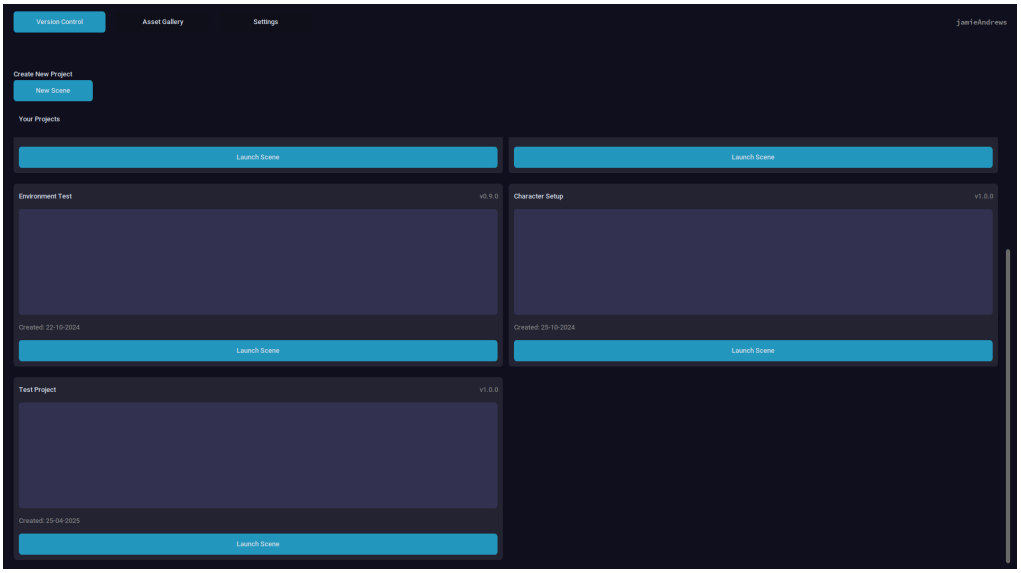
- **Hardware Requirement:** Requires a dedicated GPU due to the nature of ray tracing and the libraries being used (Rust renderer, potentially PyThreeJS). Using cloud computing like mentioned earlier would solve this problem but is expensive to set up and also requires the user to have an internet connection.

Usability Features

Several usability features were planned or partially implemented:

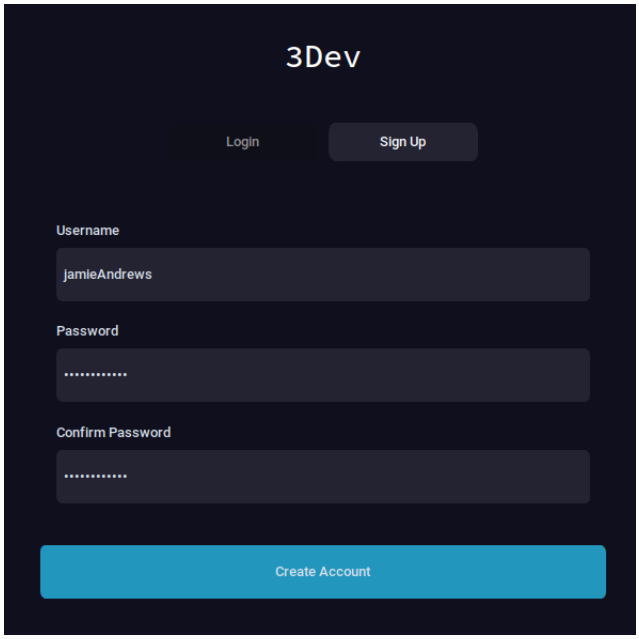
- **Intuitive UI Navigation:** Partially Successful. The sign-in process is clear, navigation between the main dashboard sections (Version Control, Asset Gallery, Settings) is straightforward and has been tested (Tests 17.1-17.3). Stakeholder feedback has indicated that the UI was easy to navigate (Sprint 3 Analysis). However, the core Scene Creator UI is missing so is obviously unusable.

Clear dashboard navigation buttons shown



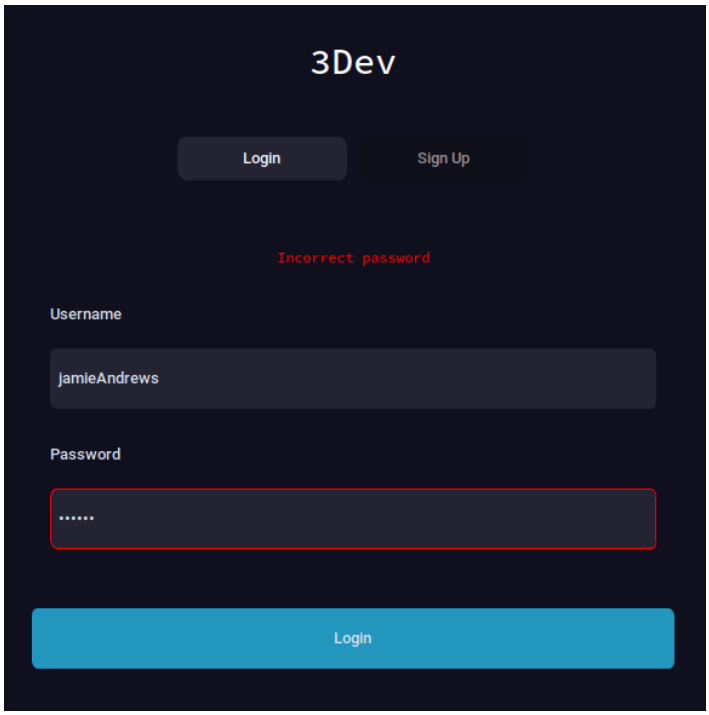
- **Consistent Theme (Blue/Dark Mode):** Mostly Successful. A consistent dark blue theme

based on research (Colour Theory section) was applied across all of the developed UI (sign-in, dashboard). This was done using hard coded colour scheming since the open source customtkinter themes were not designed to a high standard. This means that options for light mode were not functional. A custom .json theme file would need to be created for both light mode and dark mode in order to remedy this in the future.



- **Stakeholder-Informed Design:** Partially Successful. Feedback from Jason directly led to improvements in the sign-in page (Sprint 2 - Programming - Sign In). However, further usability testing with stakeholders on the full workflow (including assets and scene preview) wasn't possible due to all the incomplete features.

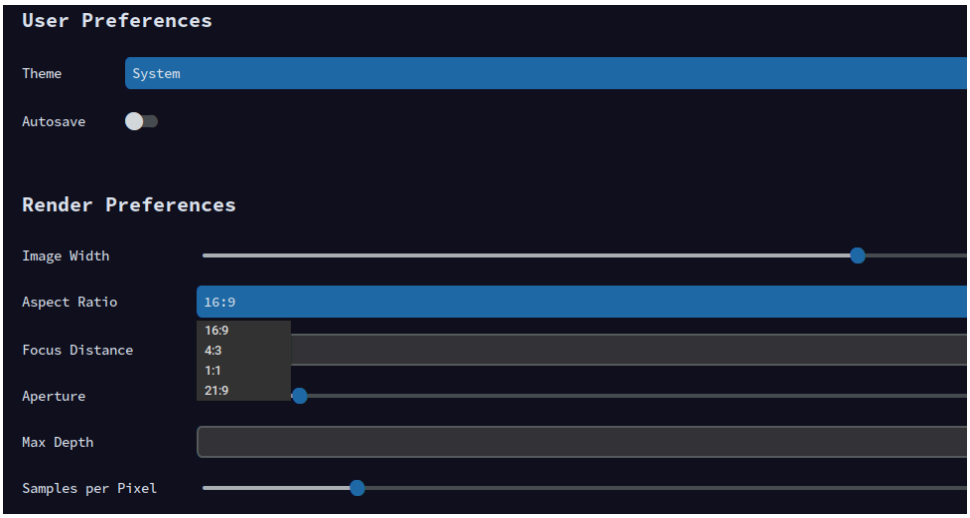
Sign Up and Settings pages shown to demonstrate theme consistency across the entire app UI.



- **Clear Forms and Feedback:** Successful. All input fields on the log-in/sign-up pages provide clear visual feedback (red/green borders, messages) upon validation (Tests 9.14, 9.21-9.26). The project creation dialogue is also clear for the same reasoning.

- **Object Manager/Scene Previews:** Not Implemented. These key usability features for the scene creation process were planned but not developed in time.

- **Sliders/Dropdowns for Settings:** Partially Implemented. Sliders and dropdowns were added to the Settings UI (e.g. for theme and render quality), aligning with my stakeholder's (Jason) preference for sliders. However, as mentioned before, not all of the settings UI elements were fully connected to the back end.



Conclusion

The project successfully covered a foundation for a high quality ray tracer capable of being easily modularised and then expanded to increase the efficiency and number of features - these changes for the future were mentioned throughout this section. This ray tracer comes alongside a

partially functional UI which clearly outlines the shell of what can become a fully developed and integrated application once the prior mentioned developments are added. However, due to time constraints shifting my focus during development, several critical components were not able to be fully or even partially completed (most notably, the Scene Creator interface). Consequently, many high-level success criteria were only partially met or not met.

The agile methodology allowed for lots of requirement flexibility, and stakeholder feedback massively impacted the UI design. Going forward, I think future sprints/iterations should be smaller so that if workload is over/underestimated for a given sprint the impact it has on the overall project completion is less. The modular structure of both the Rust and Python code provides a solid foundation for future development from myself or other developers, now that the code has been fully documented using comments throughout. Addressing the key limitations after completing the unimplemented features would definitely be the primary focus for turning this project into a fully functional application which is able to meet the original stakeholder requirements and success criteria.