## Introduction

Good morning everyone, this presentation will cover my progress thus far in creating a voxel rendering engine.

## Circumstances

I’d like to preface this presentation with a summary of the circumstances surrounding this project. This topic has only been the focus of the thesis since Week 10. Previously, an industry partnership had been arranged with Airbus which sadly feel through as they were unable to find a representative to supervise on their end. Fortunately, our QUT supervisor, Professor Clinton Fookes, has been very accommodating towards this series of events so late in the semester. Consequently, my current project is a student-led idea which I pitched as an alternative; more feasible topic due to the greatly reduced time frame which has been a significant challenge.

## Outline

So, the goal of this thesis project is to create a voxel rendering engine. A voxel is a cell of a 3D grid where each cell stores information about the density at its location in space. Essentially a 3D pixel. Due to its innate ability to store this density data, it is commonly used in medical imaging as an output data structure for scans such as MRI’s and CT’s. Another use for voxels is in video games, where they can be used to implement destructible terrain. This thesis aims to create a program that can render these objects interactively in real time with lighting and shading effects.

## Language Choice

Rendering is generally a very computationally expensive process no matter the algorithm that is being used. This is due to the sheer amount of data that needs to be created to produce an image of a usable size. Most displays these days output images at a minimum of 1080p meaning 1920 pixels wide by 1080 pixels high. For a renderer creating one of these images, this alone is already 2 million colour calculations that are likely being performed at least 30 times per second in a real-time application. To have a chance at meeting these speed requirements, I chose Rust to make this application. It’s a modern, fast, memory-safe, low-level language with performance comparable to C although with a significantly more pleasant developer experience. It has an active community, wide array of high-quality packages readily available along with a wealth of educational resources to aid in development.

## WGPU

To accomplish this project, we will be taking advantage of the GPU, which is able to process many pixels in parallel, thus improving computation times per frame by orders of magnitude. I initially opted to use WGPU for this project which is a Rust library that implements the WebGPU API, meaning it exposes a single set of methods that give us the ability to run natively in Vulkan, Metal, DirectX, and OpenGL graphics environments. To begin drawing content to the screen I followed the Learn WGPU tutorial which covered the fundamentals of the low-level library such as creating windows, event loops, adapters, and surfaces. These abstractions of the GPU pipeline where all required just to pass a single set of triangle vertices to the GPU, which was quite cumbersome and hence, spoiler alter, would later be replaced by a different system.

## Ray Marching

In the meantime, get something more appealing than a triangle drawn to the screen, I needed to become more familiar with the semantics of WGSL, the shading language used by WGPU. Having experience with the older OpenGL Shading Language, I was interested in figuring out the similarities and differences so I could apply my existing knowledge to this new system. To do this, my idea was to implement a basic but familiar ray marching render. A benefit of this was that the fundamentals of ray marching are quite like that of voxel rendering so I was hopeful some code, as well as knowledge, would be transferable. Discussing the technical details of ray marching, the whole premise is based on a function that returns the distance to the closest object in the scene given any sample point. This function is referred to as an SDF, meaning signed distance field. For each pixel, a ray is sent out at an angle determined by the field of view. The SDF is then sampled to determine the minimum safe distance the ray can be marched before intersecting with an object. This process is repeated until the SDF returns a value less than some threshold, indicating an object has been hit, or reaches some set number of maximum iterations indicating the opposite. The process described here was implemented in the fragment shader of our graphics pipeline. For each pixel, if a hit was detected Phong shading was calculated which allows the sphere here to be visualised with approximated diffuse lighting and specular highlights.

## Bevy

So, this all looks good, I was now familiar with WGSL and was confident I could build upon this knowledge. Although coming back to the complexity of the GPU pipeline, my concern was that as our application grows more complex, these more manual processes would take more and more time from actual engine development. To combat this, I decided to make the switch to a very popular Rust game engine called Bevy. Even early on the benefits of this switch would be apparent, allowing for the removal of all the WGPU boilerplate code. Very conveniently, Bevy comes with plugins for window creation, event loops, and rendering primitives, out of the box. It also has a system in place for writing custom shaders for objects which we will be using extensively. With less than 50 lines of code using the Bevy entity component system, we can create the basic 3D scene as shown here as opposed to the hundreds required previously to draw a single quad using WGPU.

Having now switch libraries, I was interested in porting over the shader code I previously wrote for WGPU. Since both libraries use the same shading language this was fortunately quite straight forward and only required changing the ray direction calculation. It was found that this could be done by accessing the points' 3D world position from within the fragment shader and then subtracting the position of the camera. The result of this "projection" of the ray marched sphere onto the cube gives the illusion of a volume as that is effectively what is being simulated. This was done after watching a tech breakdown video by the Tuxedo Labs team on the technical aspects of their game Teardown. I saw how they used cube meshes to render voxel volumes. This was also a reason behind the switch to Bevy as with the new system, it would be much easier to create many cube meshes if required in the future. This could allow for a scene to be composed of multiple voxel volumes.

## Risk Management

Thankfully, being a software application, there is a negligible chance of physical risks associated with this project. The primary risk to a software-based project such as this is data loss, where data stored in a single place can be susceptible to loss from hardware failure or by misplacing the physical device. To combat this, Git version control has been used to both keep track of incremental program changes on a line-by-line basis, as well as keep a cloud-based backup through the GitHub service.

## Sustainable Development Principles

Sustainability is another key aspect that must be taken into account when undertaking any engineering task. Being a small software-based project it is fortunate that this had very limited impact in terms of environmental sustainability. The one main impact it does have however is the power consumption of the hardware it is run on. In an attempt to minimise this, developers such as myself must strive to write efficient code that effectively uses compute cycles. So far, this has been implemented in this project by utilising the GPU for per-pixel parallel computing. Another important aspect of sustainability when completing an engineering project is being able to sustain motivation and development drive. Burnout is a common issue within the industry and seeing that development is conducted in a motivationally sustainable manner is crucial to its mitigation. Throughout development, this was achieved by taking manageable step sizes towards the goal. It would be unrealistic to start working with a new language, library, and concept, and expect to not get overwhelmed. This is the reason I started with the familiar concept of ray marching, to get my footing with both the language and the library being used. From here I have begun pivoting towards implementing the voxel features and have now become acquainted with these new technologies in a more friendly environment.

## Questions

So, this brings us to the current state of the project. Thank you for listening and do you have any questions?