## Introduction to 3D Graphics Workshop [Slide 1]

- Welcome
- Who has done graphics programming before, visualisations etc...

## Why? [Slide 2]

- Graphics programming is an incredibly rewarding area in computer science.
- Deepen your understanding of mathematics, physics, algorithms, computer hardware, optimisations, visual art, game development, tooling, and more.
- You can learn hard things, you just have to start small!

## Black Box [Slide 3]

- In software, a black box is a metaphorical representation of some component or system.
- We pass in some input, the black box does its thing and produces some output.
- To truely understand a black box we need to dig deep into the system.

## Black Box [Slide 4]

- The high-level black box in 3D graphics looks something like this.
- Our input is data representing shapes, the transformations applied to the shapes (like rotations, scale, position, etc...).
- This input passes through some black box.
- Resulting in our output, pixels being rendered to a canvas in a way the we expect.

## From Black Box -> White Box [Slide 5]

- Understanding how systems work requires us to dig deep, figure out the steps/implementation required to have a clear view of what actually happens inside the black box.
- My goal for this workshop is to demistify the black box of 3D graphics.
- But first lets think about the problem we are trying to solve.

## The Problem [Slide 6]

- Screens are 2D -> flat
- We want 3D -> objects have depth
- The Challenge (The Black Box) -> how is the perception of depth (3D) achieved on a 2D screens.

## What Are We Doing Today? [Slide 7]

- We will learn from the ground up how can we draw basic shapes and move them (1D), how we can move from screen space (pixel based space) to some world space (scaled space that is easier for us to work with).
- Learn about foundational mathematics used throughout 3D grpahics, vectors and matricies. Learn about transformations and learn how to draw more complex shapes (2D).
- And finally we will learn a few more concepts with matricies to make 3D possible.

## Don't be afraid of the C [Slide 8]

- You may have heard of the C programming language before; some may hate it because of CAB202; however, it's actually quite a simple language, also it's super portable and has been battle tested for over 50 years!
- You don't need a deep understanding of C to read the code shown during this workshop.
- If you have experience with any other language you should be able to follow.
- I will explain any C specific techniques used, so rest assured.
- To ensure that this workshop isn't a multi-day exercise, I will be using a 3rd party library to make creating a window, and drawing pixels.
- If you haven't heard of Raylib before, its a simple library used to do video game programming.
- Although raylib already has the ability to do 3D, I won't be using any of that functionality as we want to see for ourselfs how 3D works.

## Setting up the environment [Slide 9]

- I will now demonstrate the setup required to run the code used in this workshop.

## Topic 1: 1D [Slide 10]

- Lets get a feel for raylib and understand how we can draw basic things on screen.
- I'm going to open up the `main.c` inside the `src/raylib_example` directory.

## Open a window and draw a circle

- What is a window, frames, drawing a circle, moving circle, clearing the background after each frame.

[Code]

## Centering our circle -> Screen space to World space

- Moving to the boiler plate code found in `src/1D`.
- Most raylib behaviour is abstracted away - for all subsequent code demos I will be using a similar base.
- `setup()` is called before our main loop - all initialization code goes here, `update()` is called per some amount of time - code that changes the state of what is drawn is placed here, and `draw()` where all of the drawing code will go.
- You may have noticed that when we draw our circle it appears in the top left hand corner of our window.
- This is because our positions at the moment are defined in something called screen space. Where the x and y positions directly relate to pixel positions.
- This is not ideal, therefore, we have to figure out a way to center our drawing (this graphics programming equivilent of centering a div).

[Whiteboard explain screen space and world space]

[Code]

- Now like before we can move our circle around and more easily see where it's going.

## What is Space [Slide 11-12]

- Help us define where things exist.
- Here are 3 examples of different types of spaces, 1D space - basically a numberline where x=0 is the center and we can tell how far something is from the origin just by increasing x or decreasing it.
- 2D space - now we can define where something is just by adding an extra variable y. (x, y).
- 3D space - one more variable z.
- Once we go past the 1D case we can use different math tools I touch on later that makes working in higher dimensions easier.

## Geometric Primitives (Shapes) [Slide 13]

- Geometric Primitives are what we draw.

[Use whiteboard for following example]

- Drawing a square, define points, connect the dots.

## Topic 2: 2D [Slide 14]

- Now lets turn up the heat a little bit and jump in to some math.

- If you have done any first year math classes at university you should be in a pretty good spot to understand what is happening. If not, don't fear, I will try explain things simply.

## Vectors [Slide 15-17]

- What we have dealt with so far is single valued variables; x
- Vectors are a helpful tool in mathematics to help us define quantities that can't be described by a single value.
- Some examples are: Displacement (aka position),
- velocity,
- acceleration,
- force,
- colors,
- etc...
- Much like single valued variables (also known as scalars) we can operate on vectors.

[Whiteboard some examples]

- addition,
- subtraction,
- magnitude,
- *normalize

## Matrices? [Slide 18]

- Similar to how vectors are a tool for representing multiple quantities at once.
- Matrices are used to represent multiple vectors at once.
- Matrices are the excel spread sheet of math. Lets get a feel for them!

## Matrices [Slide 19]

- Rows and columns

## Transformations [Slide 20]

- The three types of transformations we will learn about today
- Translations, Scaling and Rotation.
- The identity matrix is analogous to multiplying a vector by 1. Nothing changes.
- We need to increase our dimensions to actually change the vector how we want.
- Translation.

[Whiteboard]

## Live Code Example 2 [Slide 21]

[Code Example 3]

- Translation
- Scaling
- Rotation (Skewing)

## Topic 3: 3D [Slide 22]

- It is now time for the final boss battle.
- The good news is that what we have covered so far is directly transferable to what we do in 3D.
- Vector operations are the same, we just add one more component to our vectors (z).
- Matrix operations are the same, we just add one more row and column (w).

## Vectors cont. [Slide 23]

- 3D vectors have the same operations as 2D ones - there are more but I won't go into that in this talk. You can ask me later about these operations.
- Just like the 2D case where we needed +1 component to our vectors and matrices we need a 4x4 matrix and 4D vectors to do opetaions on 3D vectors with matrices.
- When I say 4D vectors, I mean 4 components, the forth component here has nothing to do with space-time.
- What use does the fourth component have? Great question. I will talk about that soon.

## Matrices cont. [Slide 24]

- Before we looked at matrix-vector operations where we could make transformations to our geometry by multiplying some matrix against them.
- We will need the same thing in our 3D case; however, we also need a way to multiply matrices against each other.

[Whiteboard]

- Matrix multiplication
- Matrix vector multiplication

## Matrices cont. [Slide 25]

- Matrix-matrix multiplication is required because we need to multiply our model matrix against a special kind of matrix called the perspective projection matrix.
- This is the magic behind most 3D graphics.

- What is the perspective projection matrix: imagine this, you're standing in front of a window, looking out at the world. The glass acts like a projection surface. Objects that are close to you look big on the glass, and objects far away look small. That is perspective.
- This is the math trick that the perspective projection matrix pulls off. I takes 3D points from the world (x, y, z coords) and squashes them down into a 2D screen space so they look like how our eyes see them.

## Matrices (Perspective Projection) [Slide 26]

- Objects in the distance will appear smaller when using a perspective projection matrix.
- if times permitting talk about each component.

1. FOV: this is your viewing angle. How much you can see the the left and right as an angle.
2. Aspect: This will strech or squash the image so it can fit your screen shape.
3. Near and far plance: decides what's the closest thing you'll draw and what's the furthest thing we can draw.
4. *The Magic Divide: this is the w component. Right when we draw after out matrices have been multiplied each component to the drawn geometry will be divided by this w value. This division gives us the perception of depth.

## Matrices (Perspective Projection) [Slide 27]

- This may look super scary but don't worry, what I have just explained is exatly how it works. Now lets get some 3D going.

## Live Code Example 3 [Slide 28]

[Code]

- The perspective projection matrix is a tool we use that takes 3D coordinates and turns them into 2D screen positions, while making far things look smaller and near things big.

## Live Code Example 4 [Slide 29]

[Code]

## Bonus Content [Slide 30]

- Showcase model loading
- Don't need to go into too much detail

## Questions [Slide 31]