

---

---

# Computer Graphics

COMP3421/9415  
2021 Term 3 Lecture 5

---

---

# What did we learn last week?

## 2D Graphics

- OpenGL Pipeline
- Textures
- Transforms
- Some ideas on how a 2D game could be made

# What are we covering today?

## 3D Graphics

- We are entering the 3rd dimension!
- 2D to 3D . . . what changes?
- 3D Objects
- Coordinate Spaces
- Making a (virtual) Camera

# 2D to 3D

# What are our current capabilities?

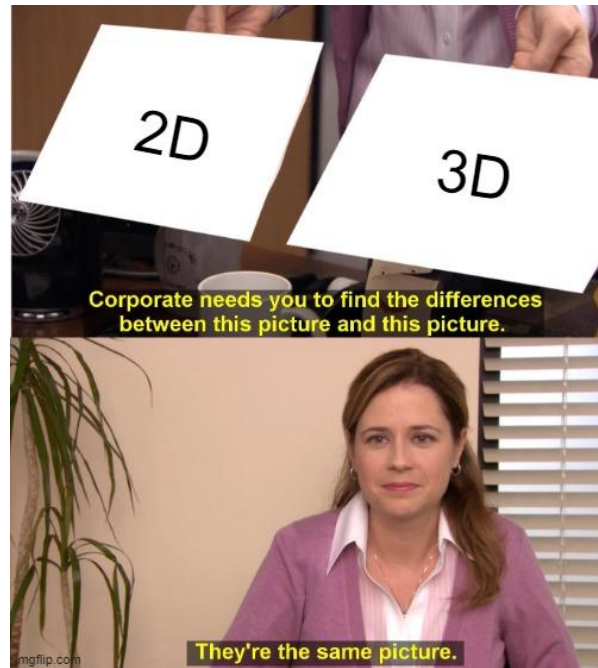
## In our 2D Graphics

- Shapes made of triangles
- Textures on objects
- Transforms

# Capabilities in 3D

What are we wanting to do in 3D?

- Shapes made of triangles
- Textures on objects
- Transforms



# Going to 3D

**We've been teaching you 3D graphics all along!**

- Only minor modifications needed
- Coordinates start to use z
- Triangles are always 2 dimensional objects . . .
- . . . but multiple triangles can make 3D objects
- Textures work with verts exactly as they do in 2D
- Transforms are going to add a dimension

# 3D Transforms

**Our Transform Matrices are adding a dimension**

- Our Vectors are now  $(x,y,z,w)$
- Our Matrices are now  $4 \times 4$



# Scale

Reasonably simple expansion into 3D

Scale x	0	0
0	Scale y	0
0	0	1

Scale x	0	0	0
0	Scale y	0	0
0	0	Scale z	0
0	0	0	1

# Translate

Reasonably simple also!

1	0	Tx
0	1	Ty
0	0	1

1	0	0	Tx
0	1	0	Ty
0	0	1	Tz
0	0	0	1

# Rotate

## Gets more interesting here

- In 3D rotation must be done AROUND a vector
- In 2D we were basically rotating around the Z axis

$\cos\theta$	$-\sin\theta$	0
$\sin\theta$	$\cos\theta$	0
0	0	1

$\cos\theta$	$-\sin\theta$	0	0
$\sin\theta$	$\cos\theta$	0	0
0	0	1	0
0	0	0	1

This row leaves the Z coordinate unaffected by the transform

This column stops the Z coordinate from affecting any others

# Rotate around other axes

We can similarly rotate around the X or Y axes

1	0	0	0
0	$\cos\theta$	$-\sin\theta$	0
0	$\sin\theta$	$\cos\theta$	0
0	0	0	1

Rotate around X

$\cos\theta$	0	$\sin\theta$	0
0	1	0	0
$-\sin\theta$	0	$\cos\theta$	0
0	0	0	1

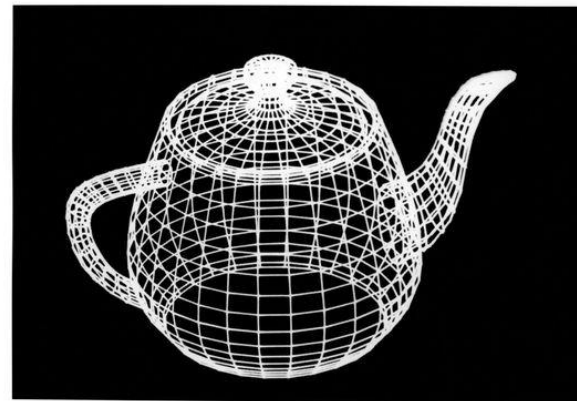
Rotate around Y

# 3D Objects

# Making 3D Objects

## Meshes of vertices

- We've already seen things like rectangles made up of two triangles
- In 3D triangles can form the outer surface of an object
- Vertices can form surfaces that wrap entirely around an object



University of Utah  
Computer Science

Image credit: School of Computing, University of Utah

# Inside vs Outside

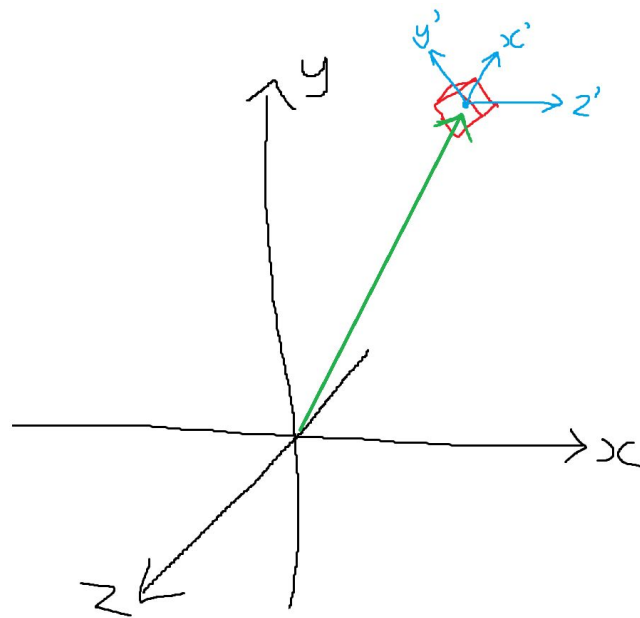
## The idea of a surface implies an inside and outside

- Triangles now have a front and a back
- Vertices go from being points in space to being positions on a surface
- These are important properties that we'll be looking at in detail later . . .

# Coordinate Spaces

Each object actually exists in its own local coordinate space

- This means each **object** actually has its own **local** origin  $(0,0,0)$
- ... which is a **point in space** in the **world** coordinates
- And its own **local**  $x, y$  and  $z$  axes
- ... which are **vectors in the world space**





# What is a transform?

**We've seen them already, but what do they represent?**

- A Transform Matrix is actually the **local** origin and axes of an object in relation to the **world** space
- When we're applying a transform, we're actually shifting an object between two coordinate systems

# Deconstructing the Transform

The Identity Matrix is the World Transform

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

The diagram shows a 4x4 matrix representing the identity transform. The first three columns are labeled 'The X axis' (red), 'The Y axis' (green), and 'The Z axis' (blue) respectively, with arrows pointing to the first, second, and third columns. The fourth column is labeled 'The origin' (black) with an arrow pointing to the fourth column. The matrix is as follows:


1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

# Deconstructing a Scale Transform

## What happens in the scale transform?

- The object's X axis is twice as "long" as the world's X axis
- This is in effect what "stretches" the object

The X axis has  
been doubled



2	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

# Deconstructing a Translate Transform

## What happens in the translate transform?

- The object has an origin of (5,6,2)
- This means that its vertices are now positioned relative to that point

1	0	0	5
0	1	0	6
0	0	1	2
0	0	0	1

The origin of the object has moved

# Composing Multiple Transforms

## Multiple Transforms together

- Retain all information from each of the transforms
- Build up a set of axes and origin for an object
- The final transform takes an object from **local** to **world** space
- It's also known as the **model matrix**

# Break Time

## The Matrix (1999)

- Speaking of important films with CG . . .
- The Matrix was rendered in Sydney by Animal Logic
- One of the Silicon Graphics Onyx machines used in the Matrix is in the lobby of the K17 building (donated by Marc Chee and others from iCinema in 2012)

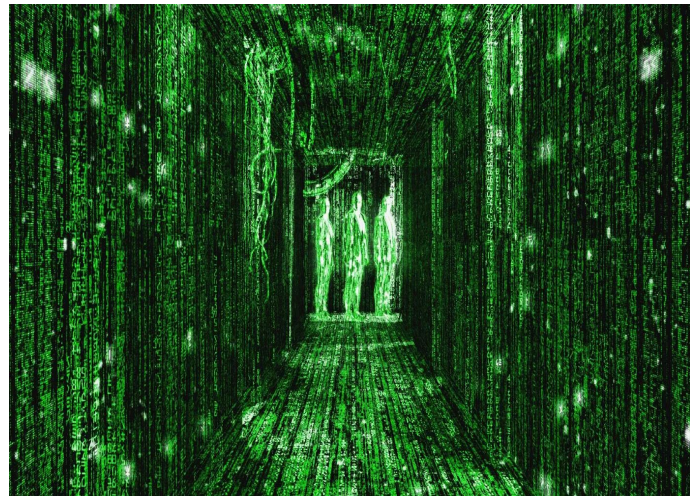


Image credit: Warner Bros Entertainment

# Cameras and Viewpoints

# Cameras as Objects in a Scene

## They exist in their own coordinate space

- So a camera will have its own transform matrix
- But it's not a 3D model, and has no vertices!
- It's more of a viewpoint that exists in the world space
- OpenGL will treat the camera as if its Z axis points from your screen to your eyes
- Using the camera transform will put all the vertices into the camera's perspective!



# Making a Camera Transform

## How do we make our camera?

- Build up the transform piece by piece

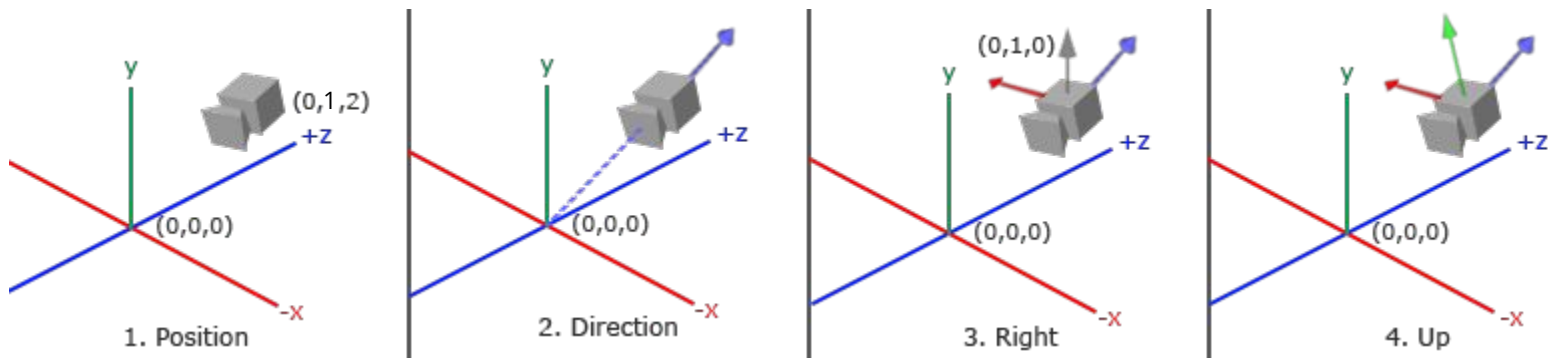


Image credit: learnopengl.com

# Camera Position

## Placing the camera in a position

- Placing something in our scene using a Translate transform
- *Let's use (0,1,2) as an example*
- *Our camera is along and just above the Z axis*

1	0	0	Px
0	1	0	Py
0	0	1	Pz
0	0	0	1

# Camera Direction

## Start building the three axes of our camera's coordinate space

- The first vector goes from where the camera is looking
- to the camera itself
- It's directly on the line the camera is looking, but aimed at the camera
- (Camera Location) - (What we're looking at)
- *In this example, we can keep it simple:*
- $(0,1,2) - (0,0,0) = (0,1,2)$

# Vectors ... Directions with Length?

We're going to want to be careful with all our vectors

- Vectors can represent points or directions
- If they represent a direction and not a distance . . .
- Then we should always **normalize** them!
- Normalize roughly means: "Make a vector length 1"
- We do this by dividing a vector by its own length
- $(0, 1, 2)$  normalized is  $(0, 1/\sqrt{5}, 2/\sqrt{5})$

# The World's Up Vector

## We have an assumption of gravity

- Humans tend to expect the camera to stay upright
- So there's always an idea of up and down in a virtual world
- We can keep this simple in most worlds by using the Y axis:
- $(0, 1, 0)$
- Is this an acceptable axis to add to the camera?

# Why have the Up Vector?

## The World's Up vector can't be trusted as an axis

- To make a set of axes, they MUST be orthogonal
- That means they're all 90 degrees from each other
- There's no guarantee the World's Up vector is 90 degrees from the Camera Direction vector
- (in fact it's incredibly unlikely!)
- But we'll use it to make one of our axes . . .

# The Right Vector

## Not the wrong vector.

- One of the axes in our camera is the one that goes to the right
- Like going across the surface of a screen from left to right
- How do we create a vector that's right angle to two other vectors?
- **Cross Product!**
- Up x Camera Direction = Right
- (remember that cross product order is important . . . right hand rule)
- $(0,1,0) \times (0,1,2) = (2,0,0)$
- *We'd normalize this to  $(1,0,0)$*

# Camera's Up Vector (or the Up Axis)

## The third axis is easy to make

- If we have two vectors, we can make a third that's orthogonal
- **Cross Product**
- Camera Direction  $\times$  Right = Up Axis
- $(0, 1, 2) \times (1, 0, 0) = (0, 2, -1)$
- *Normalized to  $(0, 2/\sqrt{5}, -1/\sqrt{5})$*



# Three Axes make a transform

## Making a Transform

- Use the vectors to make a matrix
- The Right Vector
- The (camera's) Up
- The Camera Direction
- This gives us all our rotation and scaling, but isn't yet using our position

Rx	Ux	Dx	0
Ry	Uy	Dy	0
Rz	Uz	Dz	0
0	0	0	1

# Combine the Camera Position with Orientation

## Inverting and Multiplying the two matrices together

- The resulting transform is known as the **LookAt** matrix
- Inverting the matrices moves the world relative to the camera

Rx	Ry	Rz	0
Ux	Uy	Uz	0
Dx	Dy	Dz	0
0	0	0	1

X

1	0	0	-Px
0	1	0	-Py
0	0	1	-Pz
0	0	0	1

# Will we need to do all this maths?

## Thanks again GLM

- The GL Maths Library has a single function for creating a LookAt matrix
- `glm::lookAt(position, target, up)`
- This function allows us to give only three vectors and will calculate the LookAt matrix for us

# What did we learn today?

## 2D to 3D

- A lot of what we knew still applies in 3D
- Some 3D Transforms
- Objects as meshes
- Transforms as their own coordinate spaces
- Making a Camera LookAt transform