Computer Graphics

COMP3421/9415
2021 Term 3 Lecture 6

What did we learn last lecture?

2D to 3D

- How our 2D skills relate to 3D
- Objects in 3D
- Coordinate Spaces and Transforms (they're the same thing!)
- Making a Camera

What are we covering today?

Cameras and Scenes

- Converting coordinate spaces into visibility
- A more dynamic camera

Corrections from last lecture

I'd gotten confused by Matrix Maths

- Look, it can happen to all of us :P
- The transform matrices as sets of axes
- I'd accidentally mixed up rows and columns
- All the slides in lecture 5 have now been updated!

$$
\begin{array}{cccc}
\text{The X axis} & \text{The Y axis} & \text{The Z axis} & \\
\left[\begin{matrix} 1 \\ 0 \\ 0 \\ 0 \end{matrix}\right. & \begin{matrix} 0 \\ 1 \\ 0 \\ 0 \end{matrix} & \begin{matrix} 0 \\ 0 \\ 1 \\ 0 \end{matrix} & \left.\begin{matrix} 0 \\ 0 \\ 0 \\ 1 \end{matrix}\right] \\
 & & & \text{The origin}
\end{array}
$$

Corrected LookAt Matrix

Correction from last lecture

- These two matrices are "inverted" so that they move the world and not the camera
- Hence the horizontal vectors in the rotation and the negative values in the translation

$$\begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Model/View/Projection

Where are we up to with cameras?

We've started seeing cameras as a Transform Matrix

- The LookAt matrix (also known as the View Matrix)
- This allows us to transform the world's vertices . . .
- . . . so that they're now relative to the camera

Let's look at different coordinate spaces

A vertex takes a journey through multiple coordinate systems

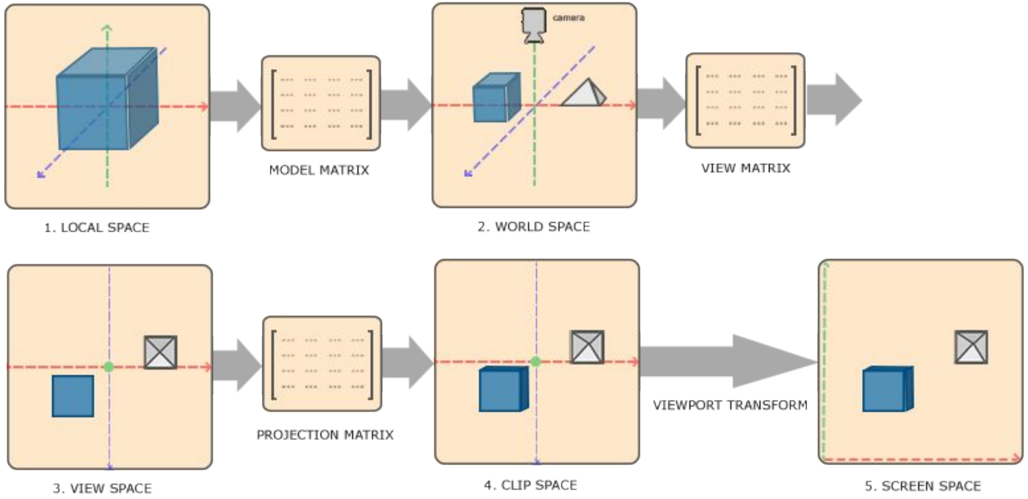


Image credit: learnopengl.com

A Vertex's Journey

From creation through to visibility

- Local Space
  - Where an object is created
- Local to World (the Model Matrix)
  - Place an object in world coordinates
  - Uses things like Scale/Rotate/Translate to position the object in the scene
- World to View
  - Uses the camera's transform (we've used LookAt to create this)
  - Coordinates are now in the camera's viewpoint

A Vertex's Journey (continued)

New transforms and coordinate spaces

- Projection to Clip Space
  - Uses the Projection Matrix
  - Figuring out the limits of what the camera can see
  - Uses Normalized Device Coordinates (-1.0 to 1.0)
  - Can also now use perspective transformation to mimic a single viewpoint
- Transform to Screen Space
  - The Viewport Transform
  - Changes our -1.0 to 1.0 into the actual pixels of the window/screen we're rendering
  - Information then goes to the rasterizer to make fragments

Projection to Clip Space

Why are we doing another transform?

- The camera's viewpoint, now known as the View Transform
  - Change the scale from World Coordinates to Normalized Device Coordinates (-1.0 to 1.0)
- Projection
  - Alter the world's coordinates so that they're a "projection"
  - We'll use Perspective or Orthographic projections
- The next step is to "clip" the vertices that we can't see
  - Any vertices outside of -1.0 to 1.0 are not visible to the camera
  - They will be discarded and will not be part of rendered fragments

The View Frustum

The Projection Matrix creates a "viewable area"

- Between -1.0 and 1.0 in three axes makes this a cube
- Forms the "viewable volume" for the camera
- This is known as the **Frustum**
- The **Near Plane** is like your screen
- The **Far Plane** is the maximum viewable distance
- Anything outside this frustum will be clipped

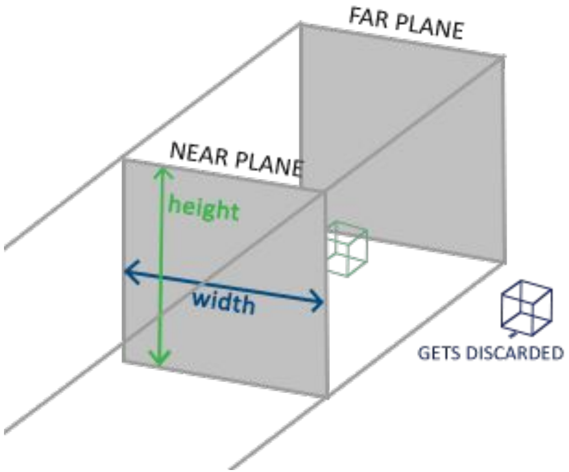


Image credit: learnopengl.com

Types of Projection

Orthographic Projection

- We've actually already been using this!
- All our 2D projects have used -1.0 to 1.0 as our coordinates
- We've been ignoring the Model/View/Projection transforms
- . . . and just working in Clip Space
- This is the same as a camera that's looking straight along the Z axis with an orthographic projection

Orthographic Projection

Looking "Square on"

- Objects don't change size based on distance
- The view frustum looks like a rectangular prism in world space

How do we see things?

Human Eyes, Real World Geometry

- We see the world from a single viewpoint
- As things get further away, they get smaller
- The idea of a "vanishing point" in the distance
- Appeared in art around the 1400s during the Italian Renaissance

Image credit: www.CGPGrey.com

Masolino da Panicale: Healing of the Cripple and Raising of Tabitha (1424)

Perspective in Graphics

Showing 3D Graphics so that our eyes believe it

- We need to replicate the human viewpoint
- The frustum for this looks interesting in world space
- It's the idea of viewing the virtual screen (Near Plane) as if from a single viewpoint
- Field of View (**FOV**) is the angle between the top and bottom of the frustum
- Aspect Ratio is the width/height of the near and far planes

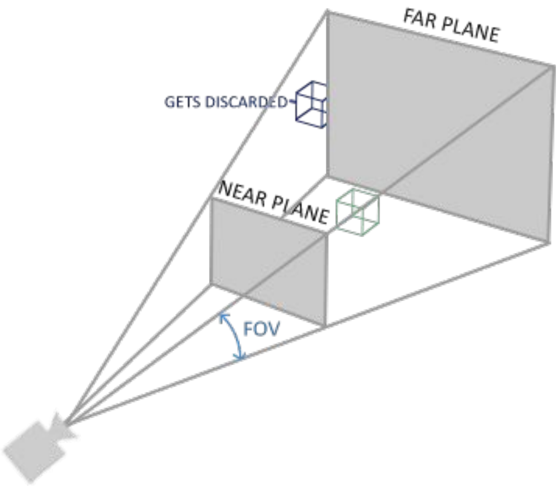


Image credit: learnopengl.com

Transforming Coordinates in Perspective

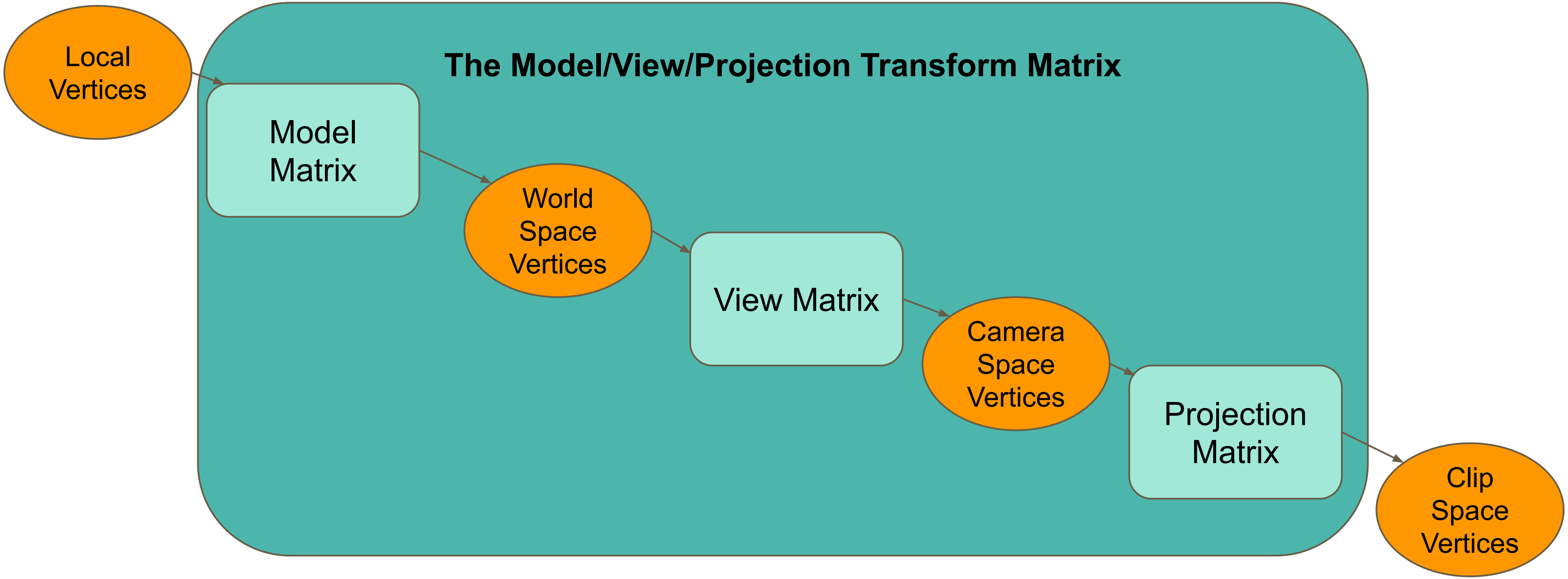
If we go from the "pyramid" frustum to a cube

- Objects closer to us will end up being enlarged
- Objects further away will be smaller

- Mathematically, we're using the w coordinate
- if we set `w = -z (with some modifications)` before applying the perspective transform
- then divide `x`, `y` and `z` by `w`
- We end up with visible coordinates in the range of -1.0 to 1.0
- We've effectively normalized our coordinates based on their distance from the camera

One Transform from Object to Screen

Multiple Matrices together can do a lot of work!



In OpenGL

We won't be building this transform matrix manually

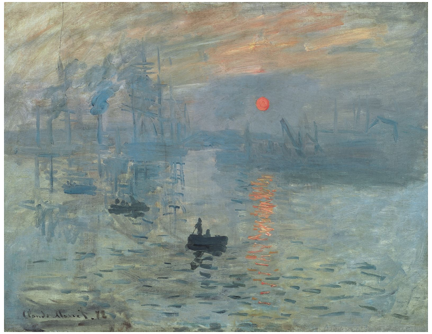
- `glm::perspective()`
- This function will take:
  - FOV
  - Aspect Ratio
  - Distance to Near Plane
  - Distance to Far Plane
- It will create a projection matrix

Break Time

An Appreciation for Technology and Art

- Perspective Projection (Renaissance 15th Century)
- Cubism, the disruption of perspective (early 1900s)
  - Picasso
- Impressionism, brush strokes predating pixels (19th Century)
  - Monet
- Colour Theory, mixing colours together (~300BC)
  - Aristotle and others along the way including Isaac Newton (1700s)

Pablo Picasso, 1910, Girl with a Mandolin (Fanny Tellier)



Claude Monet, Impression, soleil levant (Impression, Sunrise), 1872

Dynamic Camera

Moving our Camera in a Scene

Cameras are the player's view into a virtual world

- It's important that we give players control in a game situation
- Letting the camera move in the scene is amazing for immersion

Image credit: id Software (edited by Marc)

What do we have so far?

Current Camera knowledge

- We can create a transform using:
  - Position
  - Look vector (also the Camera Direction vector)
  - Up vector
- We know we can recreate this transform very quickly with new information

The Render Loop

While(true) {render}

- You may have noticed a while loop in our code
- It runs for every "frame" that is displayed on your screen
- Each time it runs, it runs the entire OpenGL pipeline
  - Calculates vertex data
  - Passes it through to fragments
  - Renders the pixel colours

Player Input

We can detect things like keyboard and mouse input

- We have some nice tools in GLFW (Graphics Library Framework)
- These can pick up keys and mouse events each frame
- We can make changes in our camera based on these
- For Example: *If 'w' is pressed, we could translate our camera towards its target by a certain amount*

How much time is there in between frames?

1/60th of a second? 1/144th of a second?

- Does this mean that a camera is going to move faster if our framerate is higher?
- Maybe we want to make sure our movement is NOT dependent on how many frames per second we are rendering

Delta Time

Make sure our render loop records time

- GLFW can give us the current time in our program
- We can record what the time was when we started rendering our last frame
- Which means we can figure out how long it took in between frames!
- This is known as delta time
- Camera speed * delta time gives us smooth motion

Rotating a Camera

Using a mouse to control where a camera is aiming

- Euler Angles: Pitch, Yaw and Roll
- Pitch rotates around the camera's x axis
- Yaw rotates around the camera's y axis
- Roll rotates around the camera's z axis

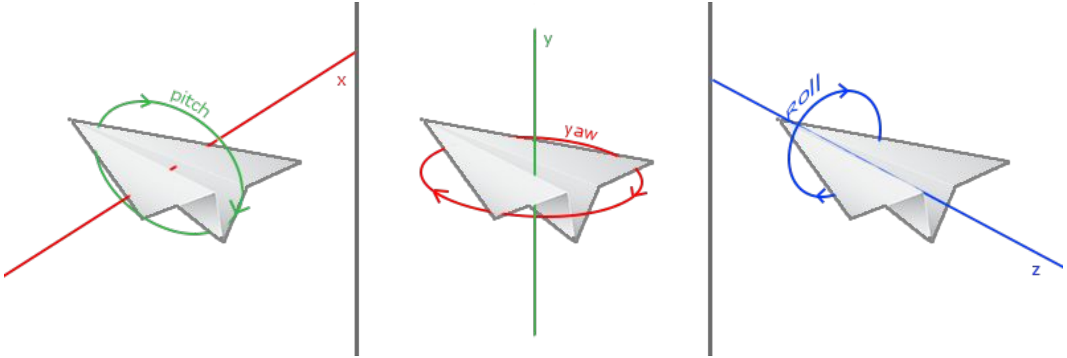


Image credit: learnopengl.com

Rotating a Camera (continued)

We're not going to be using roll (we let our up vector always stay up)

- Mouse input delta
  - Where was the mouse last frame?
  - Where is it now?
- Mouse input delta is in two dimensions
  - x relates to yaw
  - y relates to pitch
- We can calculate a new Look Vector by rotating the previous Look Vector based on the changes in the mouse input

Camera Control

Each frame . . .

- Detect the time between frames
- Detect user input
- Calculate how far the camera should move
- Calculate how much it should rotate
- Generate a new camera transform
- Pass this information to the renderer!

What did we learn today?

More details about 3D Graphics

- Model/View/Projection
  - One transform to go from local object to device coordinates
- Camera Control
  - Updating an object per frame based on player input