# COMP30019 Graphics and Interaction
# Input

Chris Ewin

Department of Computing and Information Systems
University of Melbourne

The University of Melbourne

# Lecture outline

# Why Touch?

- Touch interfaces are increasingly becoming the primary method of interaction on mobile devices.
- Microsoft requires that Windows Store apps are designed 'touch first'.
- That is, that the interaction experience is optimized for touch (but also functions on other devices).

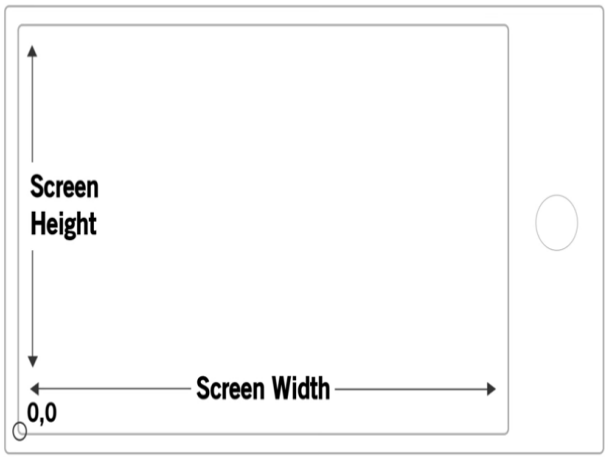
Specifically, Microsoft suggests that you:

- Design applications with touch interaction as the primary expected input method.
- Provide visual feedback for interactions of all types (touch, pen, stylus, mouse, etc.)
- Optimize targeting by adjusting touch target size, contact geometry, scrubbing and rocking.
- Optimize accuracy through the use of snap points and directional "rails".
- Provide tooltips and handles to help improve touch accuracy for tightly packed UI items.
- Don't use timed interactions whenever possible (example of appropriate use: touch and hold).
- Don't use the number of fingers used to distinguish the manipulation whenever possible.

# Touch Input

- Unity features an **Input.GetTouch** method, which allows touchscreen inputs to be processed
- This returns an array of **Touch** structs, each of which represents a single finger interacting with the screen
- Each **Touch** instance contains a:

- **position** representing the location of the finger in screen coordinates
- **deltaPosition** representing the change in position since the previous frame
- **tapCount** to distinguish multi-tap gestures
- **phase** to determine whether a particular touch has begun, ended, etc...
- **fingerID** to relate Touches between frames



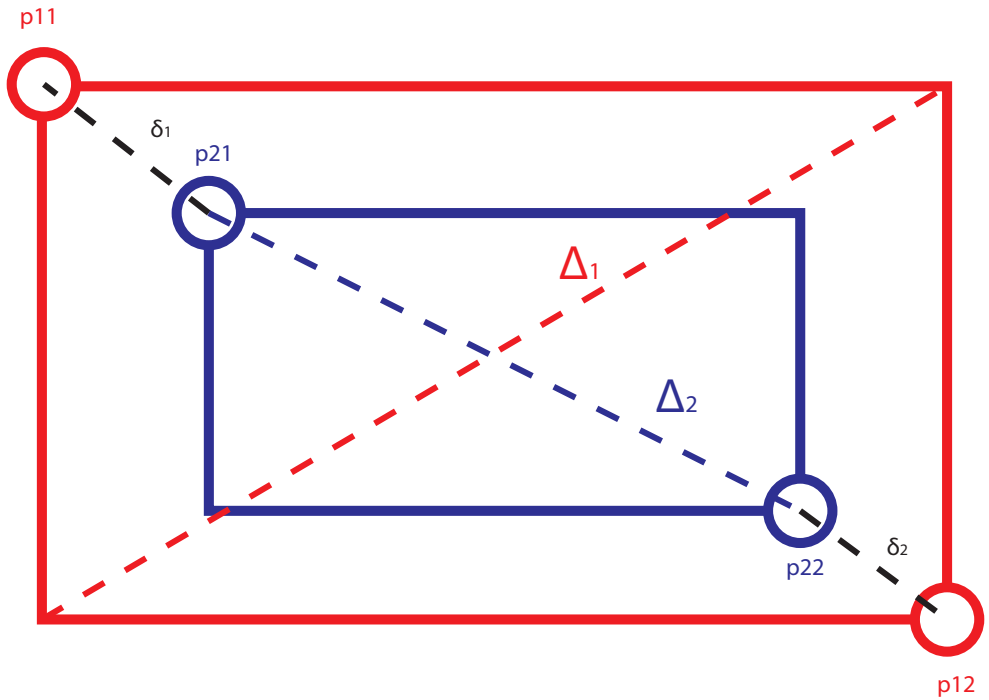
# Touch Example

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class TouchTest : MonoBehaviour
5  {
6      void Update ()
7      {
8          Touch myTouch = Input.GetTouch(0);
9
10         Touch[] myTouches = Input.touches;
11         for(int i = 0; i < Input.touchCount; i++)
12         {
13             //Do something with the touches
14         }
15     }
```

Source: Unity

# Pinch & Zoom

More complex gestures such as 'pinch & zoom' can be calculated using the different **Touch** instances.




$$p_{11} = p_{21} - \delta_1 \qquad \Delta_1 = |p_{12} - p_{11}|$$

$$p_{21} = p_{22} - \delta_2 \qquad \Delta_2 = |p_{22} - p_{21}|$$

$$fov = fov * (\Delta_1 - \Delta_2)$$

# Other Gestures

Many platforms implement their own Gesture Recognizers, designed to handle these situations. Microsoft's Windows.UI.Input, for example, provides:

- Tap
- DoubleTap
- Hold
- Drag
- ManipulationTranslate
- ManipulationRotate
- ManipulationScale
- And many others

Other platforms have similar functionality. For cross-platform support, Unity chooses not to make use of these, however third party assets exist to perform similar functionality.

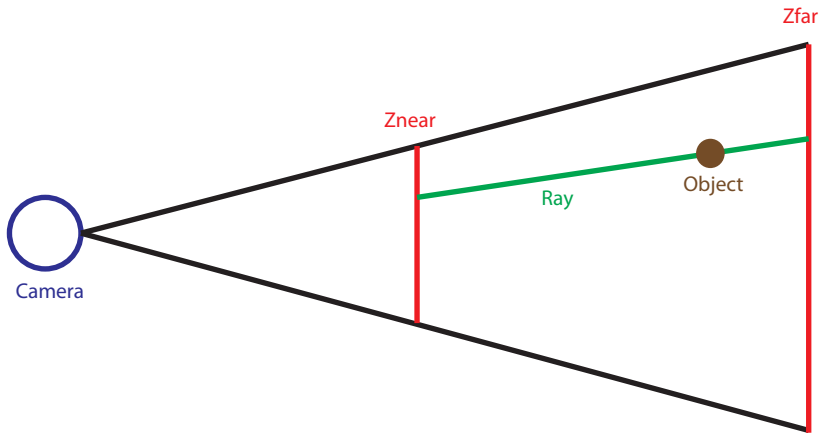
# Picking

When tapping on the screen, we often wish to calculate which game object(s) we have tapped on.

**Problem 1:** Our 'tap' is calculated in screen co-ordinates, our objects are represented in object co-ordinates!

**Problem 2:** There are an infinite number of 3D points corresponding to our 2D tap!

**Solution:**




- Cast a ray from $(X, Y, Z_{near})$ to $(X, Y, Z_{far})$ in screen space.
- Convert the ray and objects to the same co-ordinate system and check for intersection with the collision volume

# Picking

This involves converting the ray from:

- Screen co-ordinates to normalized device coordinates (NDC) (between 0 and 1)
- NDC to camera space (multiplying by $P^{-1}$)
- Camera space to world space (multiplying by $V^{-1}$)
- World space to object space (multiplying by $M^{-1}$) (or equivalently, converting objects in the scene to World co-ordinates and comparing with the ray in World space)
- Checking for intersection with the collision volume

# Picking

Fortunately, Unity provides us with an easy way to do this:

- **Camera.ScreenPointToRay(position)** creates a ray from an $(X, Y)$ co-ordinate in screen space ($Z$ is ignored) and converts back to world space.
- **Physics.Raycast(ray, hit)** returns the collider that the ray first impacts

# Sensors

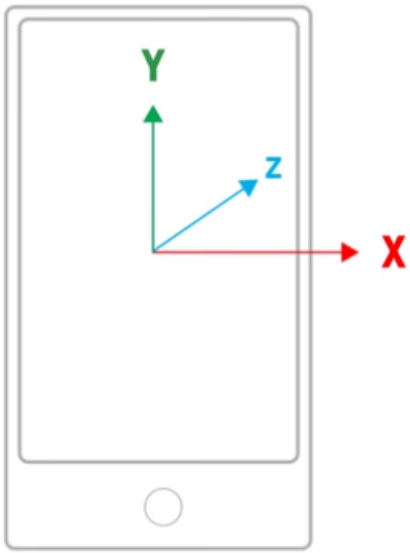
Most mobile devices will include a number of sensors that can be used as game inputs. In the case of the Surface Pro 4s that we've been using, these include

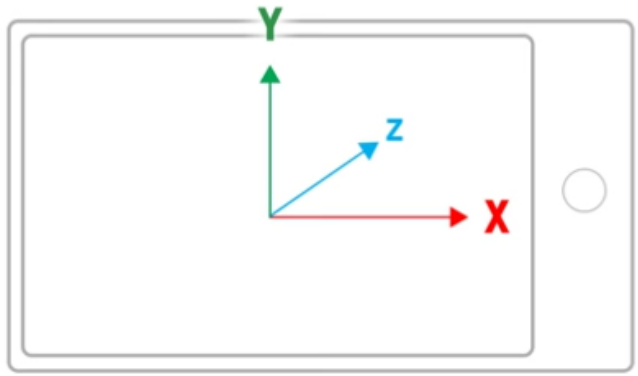
- Ambient Light Sensor
- Accelerometer
- Gyroscope (for measuring angular velocity)

Other mobile devices may include a number of other sensors, such as GPS, proximity sensors, compass, and others. It is important to check for the presence of sensors before attempting to use their input.

# Accelerometers

Probably the most useful of these sensors (for gameplay) is the accelerometer. This measures acceleration along the three principal axis. Unity will handle changes in orientation for you, but other frameworks you use may not do so.




**Question:** What will be the readings when the device is at rest?
**Answer:** `http://www.digikey.com/en/articles/techzone/2011/may/using-an-accelerometer-for-inclination-sensing`