CE318: High-level Games Development

Lecture 2: 3D Games and User Input

Diego Perez

dperez@essex.ac.uk
Office 3A.527

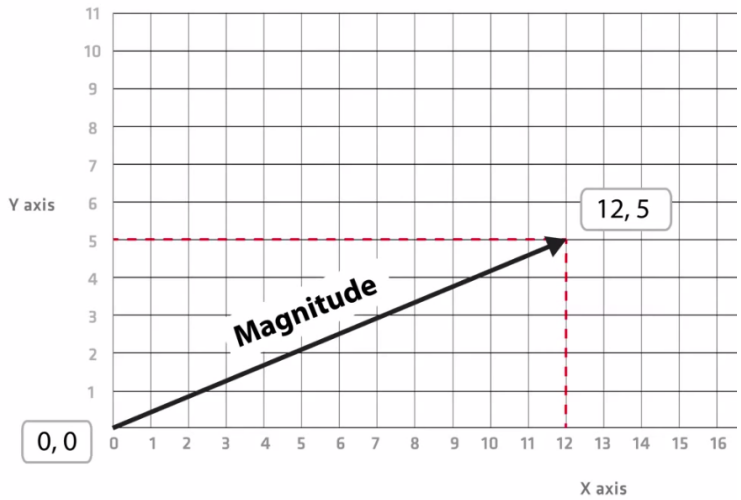
2016/17

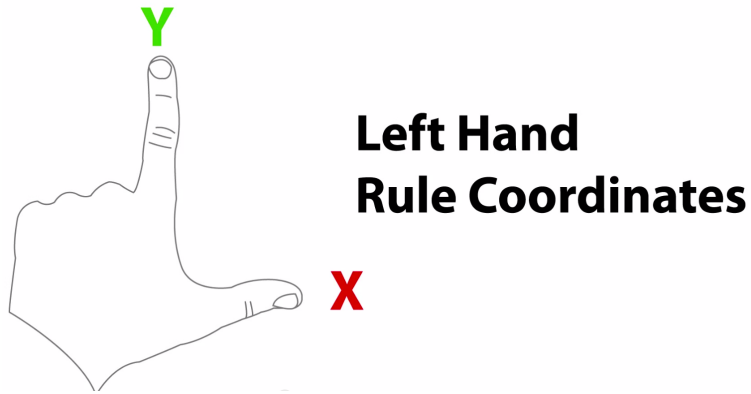
Outline

Outline

2D Vectors

A vector is a geometric object that has **magnitude** (or **length**) and **direction**.





Vectors can be used to represent a **point in space** or a **direction with a magnitude**. The magnitude of a vector can be determined by the `magnitude` attribute. 2D Vectors in Unity have many properties and methods, and all of them can be consulted in the reference:

`http://docs.unity3d.com/ScriptReference/Vector2.html`

2D Vectors & Basic Geometry

2D vectors are composed of two components, (x, y). For vectors starting at the origin, these represent the distance from $(0, 0)$ along the X and Y axis.

```
1 Vector2 vec = new Vector2(1,2);
2 int x = vec.x; //1
3 int y = vec.y; //2
```

The length (or magnitude) of a vector is determined by its norm, calculated as $magnitude = \|vec\| = \sqrt{x^2 + y^2}$. In Unity, the magnitude is available through `Vector2.magnitude` attribute.

```
1 int magnitude = vec.magnitude; // sqrt(5) = 2.236...
```

Example: a vector could represent the speed at which an object moves. If the vector is $(2, 5)$, the object is moving 2 units per second in the X axis, and 5 units in the Y axis. The magnitude of the vector $\sqrt{2^2 + 5^2} = \sqrt{29} = 5.385$ is the linear speed of the object.

Operations on Vectors (1/5)

Addition:

$$(1,3) + (2,2) = (3,5)$$

$$a + b \equiv b + a$$

Scalar multiplication:

$$x(2,3) = (2,3)x = (2x, 3x)$$

$$2(2,3) = (4,6)$$

Dot product:

$$a \cdot b = b \cdot a = \sum_{i=1}^{n} a_i b_i$$

$$a \cdot b = |a||b| \cos(\theta)$$

Subtraction:

$$(4,3) - (3,1) = (1,2)$$

$$a - b \not\equiv b - a$$

Negation:

$$a = (3,2), -a = (-3,-2)$$

$$b = (2,-2), -b = (-2,2)$$

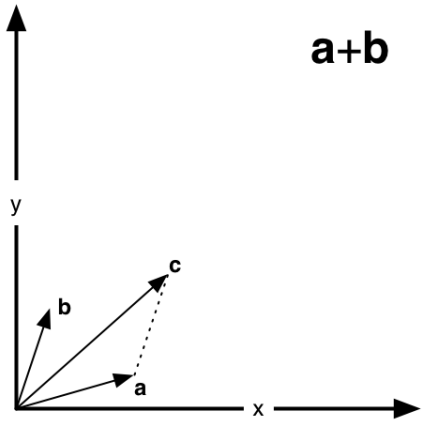
Cross product:

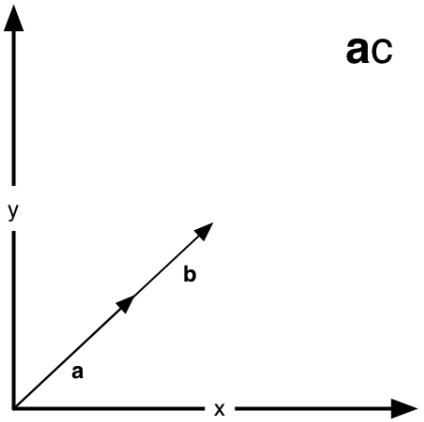
$$a \times b = |a||b| \sin(\theta) n$$

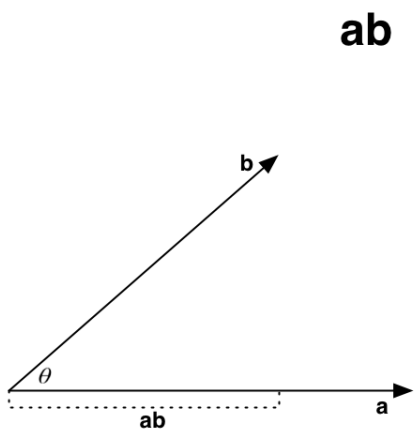
$$a \times b \equiv -(b \times a)$$

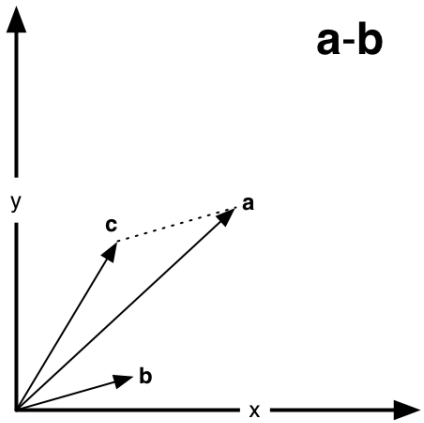
n is the unit vector perpendicular to the plane containing a and b.

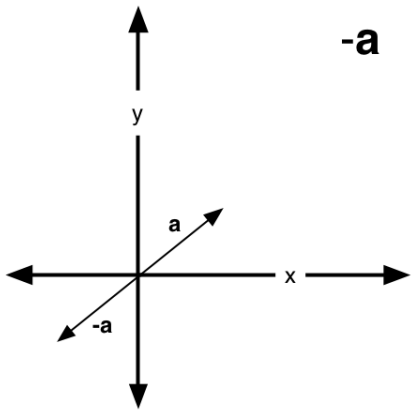
Operations on Vectors (2/5)

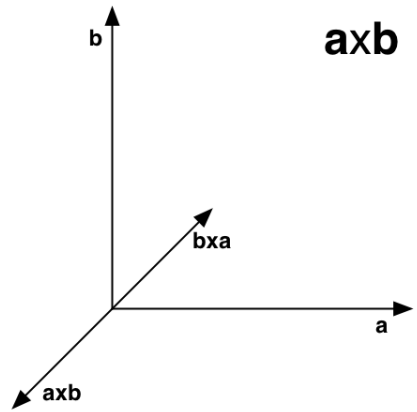












Operations on Vectors (3/5)

In Unity, you can use the normal operators for adding or subtracting to vectors, and also for multiplying or dividing a vector by a number:

```
1 Vector2 a = new Vector2(1,2);
2 Vector2 b = new Vector2(3,4);
3 int val = 2;
4
5 Vector2 sum = a + b; //or a-b
6 Vector2 mult = a * val; //or a/val
```

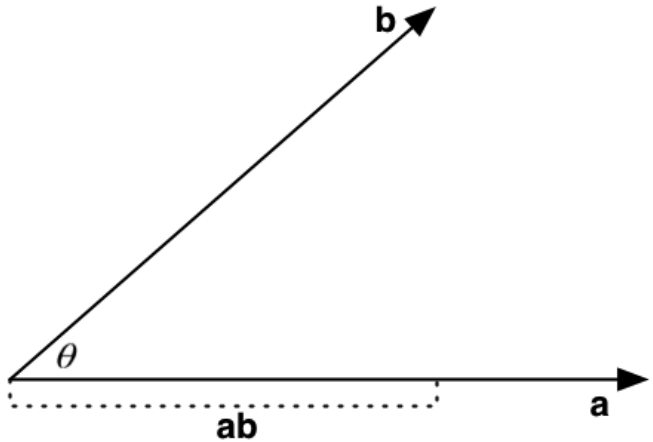
A **unit vector** is any vector with a length of 1; normally unit vectors are used simply to indicate direction. A vector of arbitrary length can be divided by its length to create a unit vector. This is known as **normalizing** a vector. A vector can be normalized in Unity by calling the function `public void Normalize`.

```
1 Vector2 a = new Vector2(1,2);
2 a.Normalize();
3 Debug.Log(a.magnitude); // =1
```

Operations on Vectors (4/5)

The **Dot Product** operation (sometimes called the inner product, or, since its result is a scalar, the scalar product) is denoted as $a \cdot b = \|a\| \times \|b\| \times cos(\theta)$, where $\|a\|$ and $\|b\|$ are the magnitudes of a and b, and θ is the angle **in radians** between these two vectors.

For normalized vectors, the dot product is 1 if they point in exactly the same direction; -1 if they point in completely opposite directions; and a number in between for other cases (e.g. Dot returns zero if vectors are perpendicular).



This is a useful operation that allows to determine the angle between two vectors. This is very useful to determine the amount of rotation needed to face certain position, angle of views, etc.

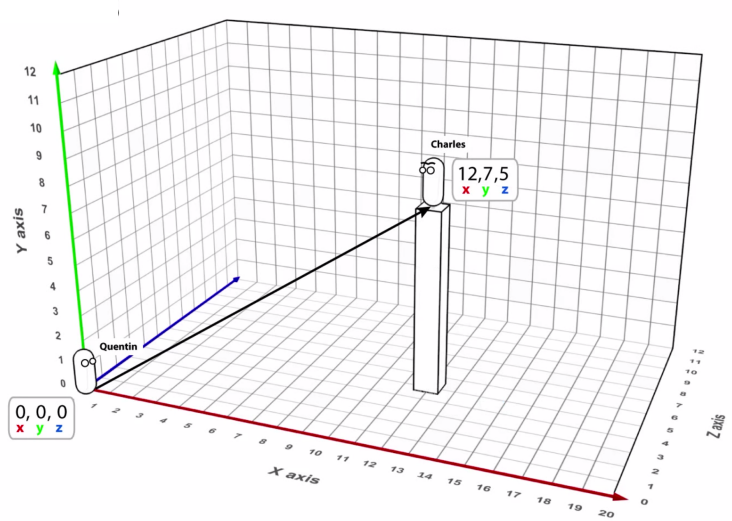
Operations on Vectors (5/5)

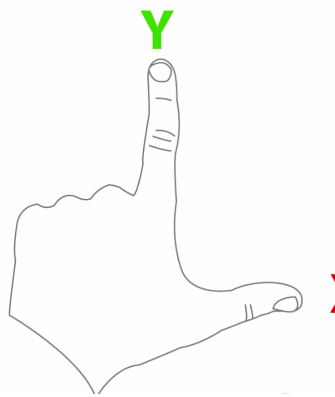
In Unity, you can use `Vector2.Dot` to calculate the dot product, calculate the angle with `Mathf.Acos()`, and multiply the radians by `Mathf.Rad2Deg` to obtain the angle in degrees.

```
1  Vector2 a = new Vector2 (0,5);
2  Vector2 b = new Vector2 (5,0);
3
4  a.Normalize (); //For consistency, normalize both vectors ...
5  b.Normalize (); //    ... before calculating its dot product.
6
7  float dotProduct = Vector2.Dot (a, b);
8  Debug.Log ("The dot product is: " + dotProduct);              // = 0
9
10 float angleRad = Mathf.Acos (dotProduct); //this is arc cos(x).
11 Debug.Log ("The angle is " + angleRad + " in radians.");   // = 1.5708
12
13 float angleDeg = angleRad * Mathf.Rad2Deg;
14 Debug.Log ("The angle is " + angleDeg + " in degrees.");   // = 90
```

3D Vectors

A 3D vector is a geometric object that has **magnitude** (or **length**) and **direction**.





Left Hand Rule Coordinates

Vectors can be used to represent a **point in space** or a **direction with a magnitude**. The magnitude of a vector can be determined by the `magnitude` attribute. 3D Vectors in Unity have many properties and methods, and all of them can be consulted in the reference:

`http://docs.unity3d.com/ScriptReference/Vector3.html`

Operations with 3D vectors (1/4)

In Unity, you can use the normal operators for adding or subtracting to vectors, and also for multiplying or dividing a vector by a number:

```
1 Vector3 a = new Vector3(1,2,-1);
2 Vector3 b = new Vector3(3,4,0);
3 int val = 2;
4
5 Vector3 sum = a + b; //or a-b
6 Vector3 mult = a * val; //or a/val
```

A **unit vector** is any vector with a length of 1; normally unit vectors are used simply to indicate direction. A vector of arbitrary length can be divided by its length to create a unit vector. This is known as **normalizing** a vector. A vector can be normalized in Unity by calling the function `public void Normalize`.

```
1 Vector3 a = new Vector3(1,2,-7);
2 a.Normalize();
3 Debug.Log(a.magnitude); // =1
```

Operations with 3D vectors (2/4)

Addition:

$$(1, 3, -1) + (2, 2, 0) = (3, 5, -1)$$

$$a + b \equiv b + a$$

Subtraction:

$$(4, 3, 5) - (3, 1, 6) = (1, 2, -1)$$

$$a - b \not\equiv b - a$$

Scalar multiplication:

$$x(2, 3, 1) = (2, 3, 1)x = (2x, 3x, 1x)$$

$$2(2, 3, 2) = (4, 6, 4)$$

Negation:

$$a = (3, 0, -1), -a = (-3, 0, 1)$$

$$b = (2, -2, 3), -b = (-2, 2, -3)$$

Dot product:

$$a \cdot b = b \cdot a = \sum_{i=1}^{n} a_i b_i$$

$$a \cdot b = |a||b| \cos(\theta)$$

Cross product:

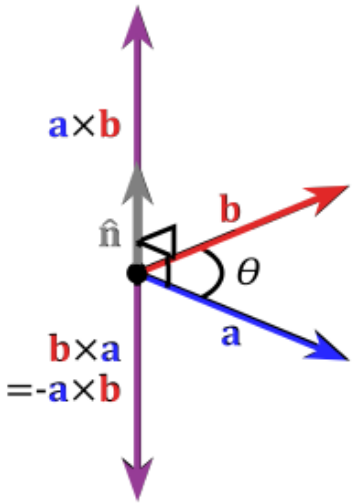
$$a \times b = |a||b| \sin(\theta) n$$

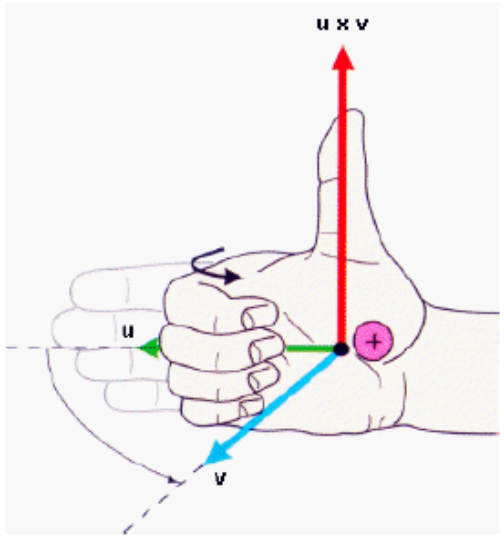
$$a \times b \equiv -(b \times a)$$

n is the unit vector perpendicular to the plane containing a and b.

Operations with 3D vectors (3/4)

The **Cross Product** operation (also called vector product) is a binary operation on two vectors in three-dimensional space, denoted as $a \times b = |a||b| \sin(\theta)n$. The cross product $a \times b$ of the vectors a and b is a vector that is perpendicular to both and therefore normal to the plane containing them. If the vectors have the same direction or one has zero length, then their cross product is zero. You can determine the direction of the result vector using the "right hand rule".

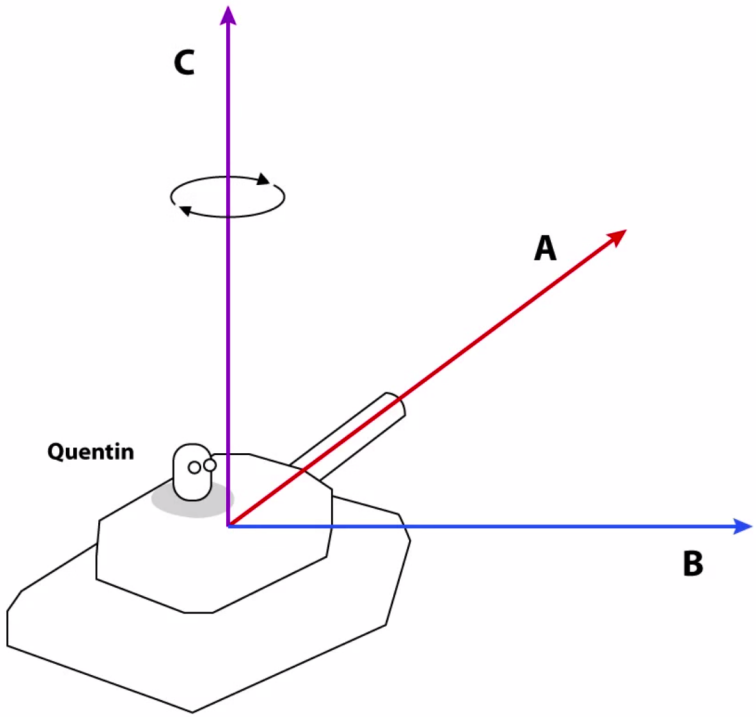





```
1 Vector3 GetNormal(Vector3 a, Vector3 b, Vector3 c) {
2     Vector3 side1 = c - a;
3     Vector3 side2 = b - a;
4     return Vector3.Cross(side1, side2).normalized;
5 }
```

Operations with 3D vectors (4/4)

An example of use of the cross product is finding the axis around which to apply torque in order to rotate a tank's turret. With the direction the turret is facing now, and the direction that it needs to face, the cross product gives the vector to rotate around.



Cross product can also be used to determine the normal of a plane (i.e. a triangle in a surface), in order to calculate directions for collisions and lighting.

Smooth Damp

```
public static Vector3 SmoothDamp(Vector3 current, Vector3 target, ref Vector3 currentVelocity
, float smoothTime, float maxSpeed = Mathf.Infinity, float deltaTime = Time.deltaTime);
```

Gradually changes a position towards a desired goal. The vector is smoothed by some spring-damper like function, which will never overshoot. A common use is for smoothing a follow camera. It returns the new position (as moved in a frame towards the goal) and sets the velocity that the object must keep.

- current: The current position.
- target: The position we are trying to reach.
- currentVelocity: Current velocity, which value is modified by the function at every call (to smoothly continue the movement in the next frame).
- smoothTime: Approximately the time it will take to reach the target. A smaller value will reach the target faster.
- maxSpeed: Optionally allows you to clamp the maximum speed.
- deltaTime: The time since the last call to this function. By default Time.deltaTime.

```
1 transform.position = Vector3.SmoothDamp(transform.position,
2                      targetPosition, ref velocity, 0.3f);
```

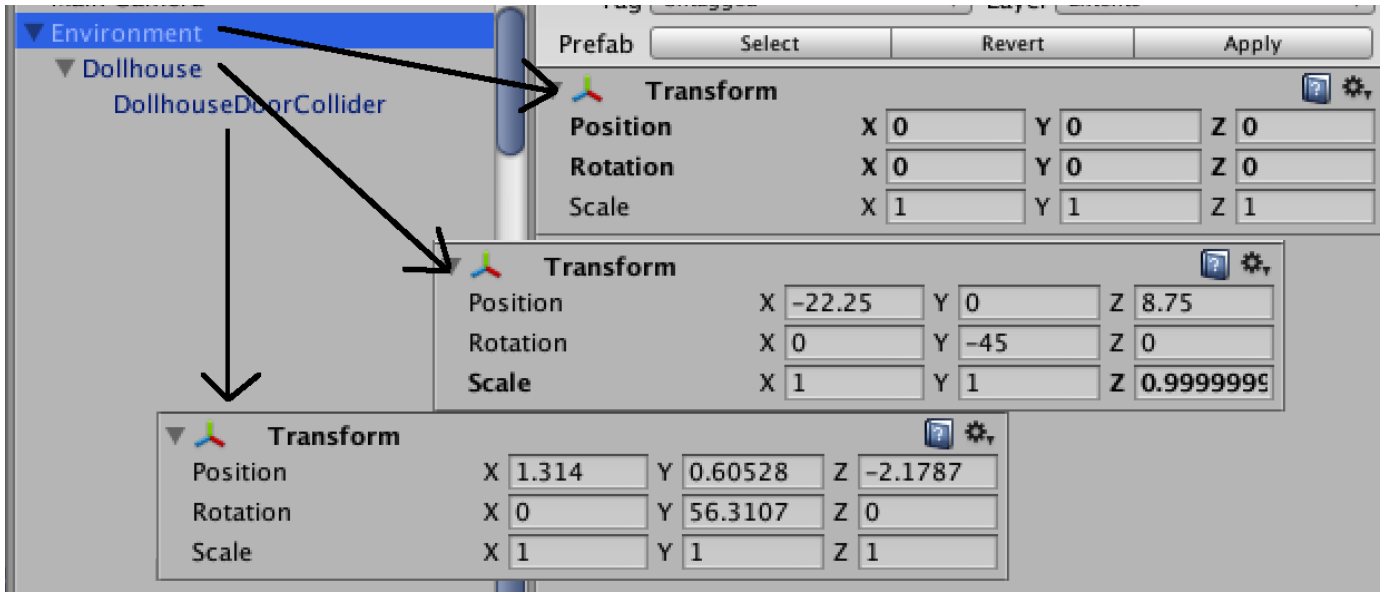
(There are equivalent functions in `Mathf` and `Vector2`)

Transforms

Every object in a scene has a Transform, that determines its position, rotation and scale.

- Position: Position of the Transform in X, Y, and Z coordinates.
- Rotation: Rotation of the Transform around the X, Y, and Z axes, measured in degrees.
- Scale: Scale of the Transform along X, Y, and Z axes. Value 1 is the original size (size at which the object was imported).

Every Transform can have a parent, which allows you to apply position, rotation and scale hierarchically.

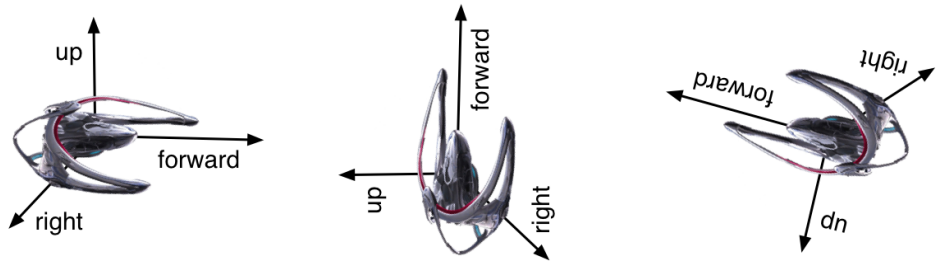


Local versus World/Global Coordinates (1/2)

The Transform component of a game object contains several vectors, such as *forward*, *up* and *right*. These vectors determine the **local** coordinate system of the game object.

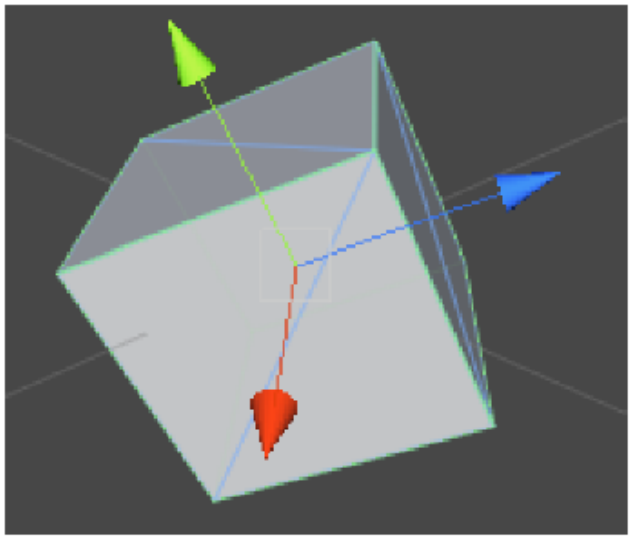
The local coordinate system (also known as *local space*, *model space* or *object space*) has all its three axes at right angles to each other: they are *orthogonal*, and can be used to know which part of the object is the front, which is the top and which is the side. Actually, they are *orthonormal*, as these three vectors are *unit* vectors.

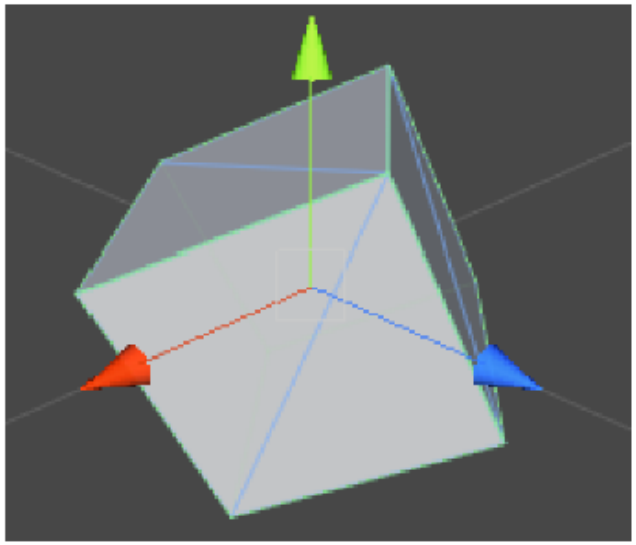
When a rotation operation is applied to modify the Transform, all axes will be rotated to reflect the change while keeping their relationship to one another.



Local versus World/Global Coordinates (2/2)*

On the other hand, the world coordinate system defines *Vector3.forward*, *Vector3.up* and *Vector3.right*, that determine the directions of positive Z, Y and X respectively, and it is a common coordinate system for all objects in the scene.





Note that the local and global coordinate systems need not be aligned. It's thus more accurate to say that you rotate around the object's forward vector, rather than the z-axis.

Transforms.Translate (1/2)

Transform.Translate:

```
public void Translate(Vector3 translation, Space relativeTo = Space.Self);
```

Moves the transform in the direction and distance of *translation*. If *relativeTo* is left out or set to `Space.Self` the movement is applied relative to the transform's local axes. (the x, y and z axes shown when selecting the object inside the Scene View.) If *relativeTo* is `Space.World` the movement is applied relative to the world coordinate system.

Examples:
A) Move the transform in the forward direction.

```
1 void Update() {
2     transform.Translate(Vector3.forward);
3 }
```

B) The previous example moves the game object forward, one unit per **frame**. But maybe one unit per frame is not precise enough (i.e. too fast). Multiply by `Time.deltaTime` to move the game object forward, one unit per **second**.

```
1 void Update() {
2     transform.Translate(Vector3.forward * Time.deltaTime);
3 }
```

Transforms.Translate (2/2)

C) In the previous examples, the speed was fixed to one unit per second/frame. We can change this:

```
1 public float speed = 5; //speed of this game object
2 void Update() {
3    //(5 units forward per second)
4    transform.Translate(Vector3.forward * Time.deltaTime * speed);
5 }
```

D) The previous example is equivalent to supplying `Space.Self` as a second parameter: the movement is relative to the local coordinates of the object. If we want to move the object in the world coordinates space (where `Vector3.Forward` $\equiv$ *Positive Z axis*), we would do:

```
1 public float speed = 5; //speed of this game object
2 void Update() {
3    //(5 units forward per second)
4    transform.Translate(Vector3.forward * Time.deltaTime * speed,
5                        Space.World);
6 }
```

Note that `Time.deltaTime` is set (by default) to $1/60 = 0.0166$. It's set to a frame rate of 60 frames per second.

Lerp

Linearly interpolates between *from* and *to* by the fraction *t*:

```
public static Vector3 Lerp(Vector3 from, Vector3 to, float t);
```

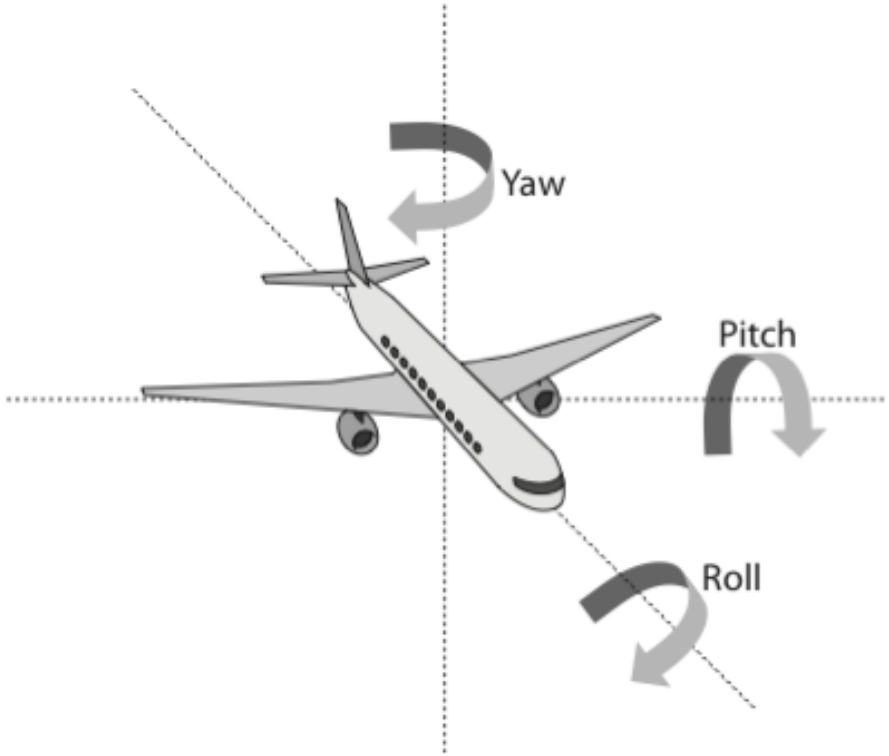
This is most commonly used to find a point some fraction of the way along a line between two endpoints (eg, to move an object gradually between those points). This fraction is clamped to the range [0...1]. When $t = 0$ returns *from*. When $t = 1$ returns *to*. When $t = 0.5$ returns the point midway between *from* and *to*.

```
1 Transform target;
2 Vector3 offset = transform.position - target.position;
3 Vector3 targetCamPos = target.position + offset;
4 float smoothing = 5f;
5
6 transform.position = Vector3.Lerp (transform.position, targetCamPos,
7                                    smoothing * Time.deltaTime);
```

(There are equivalent functions in `Mathf`, `Vector2`, `Color` and `Material`)

Rotations in 3D

There are three different rotations that can be done around a 3D point:



Each one of these rotates around a specific vector:

- Pitch: rotation around X axis.
- Yaw: rotation around the Y axis.
- Roll: rotation around Z axis.

Transforms.Rotate

Transform.Rotate:

```
public void Rotate(Vector3 axis, float angle, Space relativeTo = Space.Self);
```

Rotates the transform around *axis* by *angle* degrees. If *relativeTo* is left out or set to `Space.Self` the rotation is applied around the transform's local axes. (The x, y and z axes shown when selecting the object inside the Scene View.) If *relativeTo* is `Space.World` the rotation is applied around the world x, y, z axes. Examples:

A) A normal rotation around the Y axis (Yaw), providing the angle:

```
1 public float turnSpeed = 15f;
2 void Update() {
3     transform.Rotate(Vector3.up, turnSpeed * Time.deltaTime);
4 }
```

B) And the rotation could be applied around the local or the world coordinates space.

```
1 public float turnSpeed = 15f;
2 void Update() {
3     transform.Rotate(Vector3.up, turnSpeed * Time.deltaTime,
4                      Space.World);
5 }
```

Transform.LookAt

Transform.LookAt: Rotates the transform so the forward vector points at *target*'s current position or *worldPosition*.

```
public void LookAt(Transform target, Vector3 worldUp = Vector3.up);

public void LookAt(Vector3 worldPosition, Vector3 worldUp = Vector3.up);
```

Example: A nice use is for targeting the camera to a position:

```
1 public class CameraLookAt : MonoBehaviour
2 {
3     public Transform target;
4
5     void Update ()
6     {
7         transform.LookAt(target);
8     }
9 }
```

Transform.localScale

Unity does not include any *Transform.Scale* function. However, it is possible to modify the field *localScale*, to modify the scale of the transform.

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class ExampleClass : MonoBehaviour {
5     void Example() {
6         transform.localScale += new Vector3(0.1F, 0, 0);
7         transform.localScale *= 1.005f;
8     }
9 }
```

Important: Remember that if you want to move an object with a collider (it will interact with physics), you should **not** use `Transform.Translate` or `Transform.Rotate`. Instead, you should use the physics functions instead (see next lecture). You should use `Transform.Translate` or `Transform.Rotate` when the object has a rigid-body marked as kinematic.

The Gimbal Lock Problem (1/2) *

Rotating a game object directly using Euler degrees is very intuitive. In a translation, a quantity on each one of the axis is given to change the position. If an object's transform position is (x, y, z) and we want to change this by moving it with $(\delta x, \delta y, \delta z)$, the final position will be $(x + \delta x, y + \delta y, z + \delta z)$. The same holds true for rotations.

Internally, the Transform holds a Matrix formed by the (local) vectors *Right*, *Up* and *Forward*. When applying a rotation in $\ominus$ (radians), the following matrix are used to multiply and change the transform matrix:

X-Rotation - Pitch:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\ominus & -\sin\ominus \\ 0 & \sin\ominus & \cos\ominus \end{bmatrix}$$

Y-Rotation - Yaw:

$$\begin{bmatrix} \cos\ominus & 0 & \sin\ominus \\ 0 & 1 & 0 \\ -\sin\ominus & 0 & \cos\ominus \end{bmatrix}$$

Z-Rotation - Roll:

$$\begin{bmatrix} \cos\ominus & -\sin\ominus & 0 \\ \sin\ominus & \cos\ominus & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This way of applying rotations works fine, but it is exposed to the **gimbal lock problem**: *the loss of one degree of freedom in a three-dimensional space that occurs when the axes of two of the three gimbals are driven into a parallel configuration.*

The Gimbal Lock Problem (2/2)

A possible set of rotations is the following combination of matrices. If $\beta = 0$:

$$R = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta \\ 0 & \sin\beta & \cos\beta \end{bmatrix} \times \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} \cos(\alpha+\gamma) & -\sin(\alpha+\gamma) & 0 \\ \sin(\alpha+\gamma) & \cos(\alpha+\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In this configuration, there are two angles available, but effectively only one angle changes the transform $(\alpha + \gamma)$. Changing the values of α and γ in the above matrix has the same effects: the rotation angle $\alpha + \gamma$ changes. The rotation axis remains in the Z direction: the last column and the last row in the matrix won't change.

For a detailed description of the gimbal lock problem (in this, and other engineering problems), check:

`https://sundaram.wordpress.com/2013/03/08/mathematical-reason-behind-gimbal-lock-in-euler-angles/`

Quaternions (1/3)

Quaternions are four dimensional vectors of real numbers that allow three operations: addition, scalar multiplication and quaternion multiplication. They are used to represent rotations. They don't suffer from gimbal lock and can easily be interpolated. Unity internally uses Quaternions to represent all rotations.

Quaternions work with four components: w, x, y and z. These components work together, and they **should never** be modified individually.

Quaternions vs. **Euler**:

- Quaternions suffer from no gimbal lock, providing an smoother (than Euler) interpolation.
- Calculations are easier in quaternions.
- Conceptually, quaternions are more difficult to grasp than Euler transformations.
- They work at different levels:
  - Quaternions represent a Delta (displacement) in a *sphere* for rotations.
  - Euler rotations are made as a chain or path of rotations.

Quaternions (2/3) *

The class `Quaternion` offers the following functionality:

Quaternion.LookRotation:

```
public static Quaternion LookRotation(Vector3 forward, Vector3 upwards = Vector3.up);
```

Creates a rotation with the specified *forward* and *upwards* directions. If used to orient a Transform, the Z axis will be aligned with forward/ and the Y axis with upwards if these vectors are orthogonal.

```
1 public class ExampleClass : MonoBehaviour {
2     public Transform target;
3     void Update() {
4         Vector3 relativePos = target.position - transform.position;
5         Quaternion rotation = Quaternion.LookRotation(relativePos);
6         transform.rotation = rotation;
7     }
8 }
```

Quaternions (3/3) *

Quaternion.Slerp:

```
public static Quaternion Slerp(Quaternion from, Quaternion to, float t);
```

Spherically interpolates between *from* and *to* by *t*.

```
1 public class ExampleClass : MonoBehaviour {
2     public Transform from;
3     public Transform to;
4     public float speed = 0.1F;
5     void Update() {
6         transform.rotation = Quaternion.Slerp(from.rotation, to.
              rotation, Time.deltaTime * speed);
7     }
8 }
```

(There is an equivalent function in `Vector3`)

Quaternion.identity:

```
public static Quaternion identity;
```

The identity rotation (Read Only). This quaternion corresponds to "no rotation". Is perfectly aligned with the world or parent axes.

```
1 transform.rotation = Quaternion.identity;
```

Outline

The Input Manager (1/2)

Unity supports the conventional types of input device used with games (keyboard, joypad, etc) but also the touchscreens and movement-sensing capabilities of mobile devices.

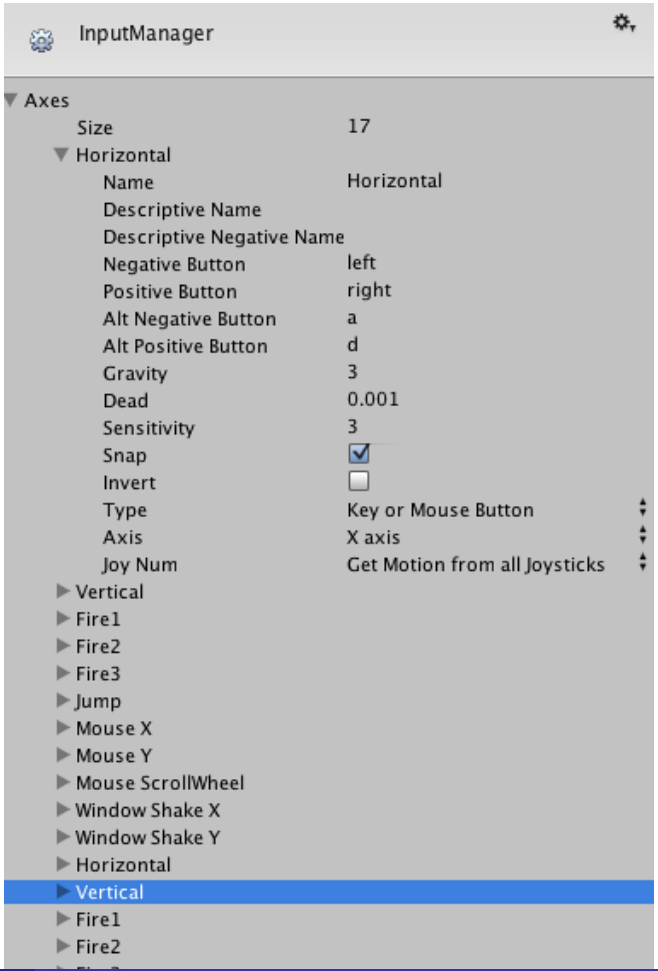
The Input Manager is where you define all the different input axes and game actions for your project.

To see the Input Manager choose: *Edit* → *Project Settings* → *Input*.

All the axes serve two purposes:

- Allows to reference by name in scripting.
- Allow the players to customize the controls.

All defined axes will be presented to the player in the game launcher, where they will see its name, detailed description, and default buttons. From here, they will have the option to change any of the buttons defined in the axes.



The Input Manager (2/2)

These are the most relevant settings that can be established for each axes:

- Name: The string that refers to the axis in the game launcher and through scripting.
- Positive Button: The button that will send a positive value to the axis.
- Descriptive Name: A detailed definition of the Positive Button function that is displayed in the game launcher.
- Negative Button: The button that will send a negative value to the axis.
- Descriptive Negative Name: A detailed definition of the Negative Button function that is displayed in the game launcher.
- Gravity, Dead, Snap and Sensitivity: Referred to `GetAxis`, see later below.
- Invert: If enabled, the positive buttons will send negative values to the axis, and vice versa.
- Type:
  - Key / Mouse Button for any kind of buttons.
  - Mouse Movement for mouse delta and scrollwheels.
  - Joystick Axis for analog joystick axes.
  - Window Movement for when the user shakes the window.
- Axis: Axis of input from the device (joystick, mouse, gamepad, etc.)

Conventional Input

Every project has the following default input axes when it is created:

- Horizontal and Vertical are mapped to w, a, s, d and the arrow keys.
- Fire1, Fire2, Fire3 are mapped to Control, Option (Alt), and Command, respectively.
- Mouse X and Mouse Y are mapped to the delta of mouse movement.

All virtual axes are accessed by their name from the scripts. You can query the current state from a script like this:

```
1 float value = Input.GetAxis ("Horizontal");
2 bool  value = Input.GetKey ("a");
```

An axis has a value between -1 and 1. The neutral position is 0. This is the case for joystick input and keyboard input.

Mouse Delta and Window Shake Delta are how much the mouse or window moved during the last frame, so it can be larger than 1 or smaller than -1.

Buttons and Keys

In Unity, the `GetKey` and `GetButton` family functions are ways of receiving input from keys or joystick buttons via Unity's `Input` class.

`GetKey` referes to specific keys that can be pressed. The class `KeyCode` allows to specify concrete keys:

```
1 bool down = Input.GetKeyDown(KeyCode.Space);
2 bool a = Input.GetKeyDown(KeyCode.A);
```

However, it is recommended to use `GetButton` instead, as this allows you to access the input controls specified in the Input Manager.

`GetKey` and `GetButton` families have three members that return a `bool` depending on the key being pressed or not.

- `Input.GetKeyDown()` / `Input.GetButtonDown()`: When a key or button is pressed, it returns `true`, only in the first frame when it is pressed.
- `Input.GetKey()` / `Input.GetButton()`: Returns `true` when the key or button is pressed, during every frame it is maintained pressed (this includes the first frame). It returns `false` in the frame the key is released.
- `Input.GetKeyUp()` / `Input.GetButtonUp()`: When a key or button is released, it returns `true`. As the down variant, it only lasts one frame.

```
1 bool jump = Input.GetButtonDown("Jump");
```

GetAxis and GetAxisRaw

`GetAxis` returns a float value between -1 and 1, and `GetAxisRaw` returns an float with the values -1, 0 and 1 only. With axes, we should consider the Positive (value 1) and Negative (value -1) buttons defined in the Input Manager.

```
1 float h = Input.GetAxis("horizontal");
2 float v = Input.GetAxis("vertical");
3 float hR = Input.GetAxisRaw("horizontal");
4 float vR = Input.GetAxisRaw("vertical");
```

There are four settings for GetAxis():

- Gravity affects how fast the value returns to 0 after the button has been released.
- Sensitivity affects how fast the value arrives at -1 and 1 after the button has been pressed.
- Dead: Any positive or negative values that are less than this number will register as zero. Useful for joysticks, so you can ignore very small amounts of joystick movement.
- Snap option (if checked) allows to return 0 if both positive and negative buttons are held simultaneously

Nice animation in this video:

`http://unity3d.com/learn/tutorials/modules/beginner/scripting/get-axis`

Mouse

Mouse input can be detected with the `GetAxis` and `GetButton` functions, as the input from the keyboard is managed.

However, Unity also allows to detect clicks on a collider or a GUI element. If this element has a script attached, the following family of functions will be called:

- `void OnMouseDown()`: called when the user has pressed the mouse button while over the GUIElement or Collider.
- `void OnMouseUp()`: when the user has released the mouse button.
- `void OnMouseEnter()`: when the mouse entered the GUIElement or Collider.
- `void OnMouseOver()`: every frame while the mouse is over the GUIElement or Collider.
- `void OnMouseExit()`: when the mouse is not any longer over the GUIElement or Collider.
- `void OnMouseDrag()`: when the user has clicked on a GUIElement or Collider and is still holding down the mouse.
- `void OnMouseUpAsButton()`: when the mouse is released over the same GUIElement or Collider as it was pressed.

Outline

Lab Preview

During this lab, you will complete the second part of the lab. Remember that you will be assessed and you will continue from where you left it last week.

Next lecture: 3D Games: Models and Physics.