InvenSense Inc.
1197 Borregas Ave., Sunnyvale, CA 94089 U.S.A.
Tel: +1 (408) 988-7339 Fax: +1 (408) 988-8104
Website: www.invensense.com

Document Number:
Revision:
Release Date:

Making Android Motion Applications Using the Unity 3D Engine

A printed copy of this document is
NOT UNDER REVISION CONTROL
unless it is dated and stamped in red ink as,
"REVISION CONTROLLED COPY."

TABLE OF CONTENTS

1. Revision History

Revision Date	Revision	Description
06/26/2012	1.0	Document created

2. Purpose

This tutorial is about the basics of designing games and applications with motion interface using the Unity 3D game engine. This tutorial will guide you through Unity 3D development from the root level by focusing on development environment, tools, concepts of motion interface, using Gyro class, and scripting. The tutorial will then demonstrate how all of these can be integrated to create a 3D motion interface design.

3. Requirements

- Unity 3D Pro version with Android development support.
- Android SDK.
- Familiarity with JavaScript or C#.

4. Introduction to Unity 3D Game Engine

Unity3D is a cross-platform 3D gaming engine for app developers to build 3D/2D games and applications. It provides visual editing options for building, importing, and using 3D models and game objects from 3D modeling tools (e.g. Maya, 3DS Max, etc.) while also providing scripting behaviors for interacting with those objects.

Starting with version 3.4, Unity 3D provides motion sensor run-time classes for Android developers to create motion based applications. In this tutorial we will teach the readers how to create an Android application that responds to device motion using the Unity 3D engine.

4.1 Getting Started with Unity 3D

The Unity 3D website provides well explained documentation and tutorials for getting started with game development in Unity 3D and also provides a detailed account for the scripting API's. Interested readers please refer to the following links:

- Learning the Interface
- Asset Workflow
- Creating Scenes
- Publishing Builds

4.2 Motion Sensor Classes in Unity 3D (Version 3.4)

Unity 3D v3.4 provides run-time classes for developers to access different motion sensors, such as Gyroscope, Compass, and Acceleration. To create a real-time motion application that instantly responds to device motion, we recommend the readers to use the "Gyroscope.attitude" variable in the Gyroscope class. The Gyroscope.attitude variable represents the 3D orientation of the device with a 4-dimensional vector called "quaternion" [5]. Unity 3D acquires the quaternion value from the Rotation Vector event that the Android Sensor Manager created using 9-axis sensor fusion (Android also provides a helper function to convert the rotation vector to quaternion). In this tutorial we will provide a simple example to demonstrate how to use Gyroscope.attitude to create a motion-based first person shooter game.

Note: In Unity 3D v3.4, the elements in the Gyroscope.attitude vector need to be rearranged to match the conventional definition of the quaternion in [5]:

```
gyro = Input.gyro; // Store the reference for Gyroscope sensor
Quaternion transQuat = Quaternion.identity;
//Adjust Unity output quaternion as per Android SensorManager
transQuat.w = gyro.attitude.x;
transQuat.x = gyro.attitude.y;
transQuat.y = gyro.attitude.z;
transQuat.z = gyro.attitude.w;
transQuat = Quaternion.Euler(90, 0, 0)*transQuat;//change axis around
```

A more detailed explanation will be provided in the following sections.

5. Making a Game App Respond to Motion

5.1 Porting Viewport Navigation Script of FPS to Motion Sensors API

This section will walk you through a practical example that builds a Unity 3D application and also illustrates the capabilities and merits of motion interface over conventional touch and UI interface. We will use the First Person Shooter example game from Unity 3D. We will create a script for the First Person Shooter game which will control the camera movement in the scene in accordance with the device motion.

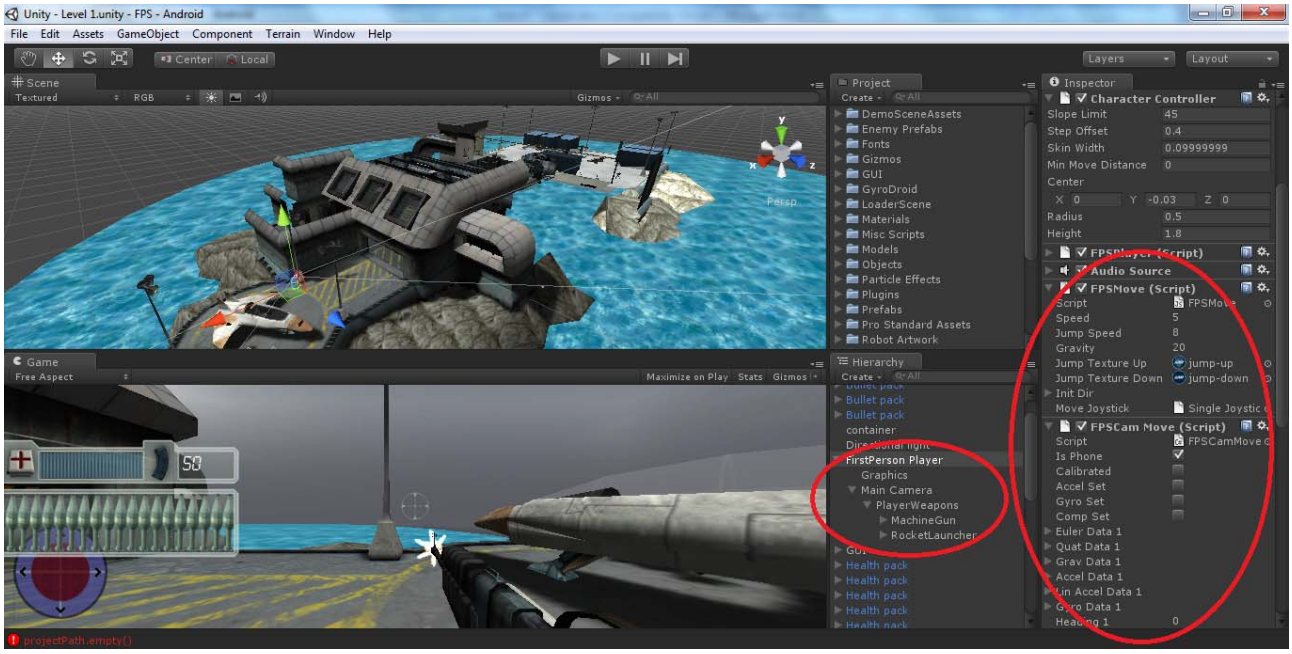


Figure 1. FPS Project Setup in Unity. The circle (left) highlights the game object hierarchy for Player, while the circle (right) is highlighting the inclusion of edited scripts.

Figure 1 shows the First Person player game object hierarchy. There are two scripts that are attached to the first person player i.e. FPSWalker and MouseLook script. The first person character in the game is an object that contains a camera at eye level to navigate through the scene and aim weapons. FPSWalker is the script that is responsible for making the character walk in a particular direction, while the MouseLook script is responsible for looking around the scene at a 360 degree angle. Replacement script for FPSWalker, which is called FPSMove, is given as an example in APPENDIX A (FPSWalker uses keyboard input, so we have to change this to use the touch joystick and the touch jump button to fit it for the Android user interface). We will port the functionality of the MouseLook script to replace it with Motion Interface.

The MouseLook script performs the following actions to look around the scene.

- Get input along the X and Y axis of the mouse and multiply it by the sensitivity constant.
- Increment the rotation variables for x and y axis respectively.
- Clamp angle to limit the rotation to a range.
- Get quaternion for rotation around the x and y axis and assign it as the rotation for character

Below is the code snippet in C# for `Update()` function in the original FPS example, which uses the mouse to control the viewing direction:

```
        void Update (){
                rotationX += Input.GetAxis("Mouse X") * sensitivityX;
                rotationY += Input.GetAxis("Mouse Y") * sensitivityY;

                rotationX = ClampAngle (rotationX, minimumX, maximumX);
                rotationY = ClampAngle (rotationY, minimumY, maximumY);

                Quaternion xQuaternion = Quaternion.AngleAxis (rotationX, Vector3.up);
                Quaternion yQuaternion = Quaternion.AngleAxis (rotationY, Vector3.left);

                transform.localRotation = originalRotation * xQuaternion * yQuaternion;
        }

        void Start (){
                // Make the rigid body not change rotation
                if (rigidbody)
                        rigidbody.freezeRotation = true;
                originalRotation = transform.localRotation;
        }

        public static float ClampAngle (float angle, float min, float max){
                if (angle < -360F)
                        angle += 360F;
                if (angle > 360F)
                        angle -= 360F;
                return Mathf.Clamp (angle, min, max);
        }
```

Update() function is called on every frame of the game, that is why we are using this function to update the rotation transform of the player. To replace the mouse control in Update() with motion control, we will use the Gyroscope class from the Unity 3D scripting API that provides access to the gyroscope. The gyroscope class provides various outputs such as rotation rate, attitude of the device etc.

The next step is to create a new Unity C# script named "FPSCamMove" to replace the MouseLook script. FPSCamMove script uses the Gyroscope class to look around in the scene. We activate the gyroscope by overriding the Start() function i.e.:

```
void Start(){
        gyro = Input.gyro; // Store the reference for Gyroscope sensor
        gyro.enabled = true; //Enable the Gyroscope sensor
}
```

The easiest way to rotate the camera around the scene is to adjust the attitude of the device in the form of quaternion to fit the sensor coordinate system of Android, and then set this quaternion as a camera rotation transform. Here is how we override the Update() function:

```
void Update(){
        if (Time.timeScale != 0){

                Quaternion transQuat = Quaternion.identity;
                //Adjust Unity output quaternion as per android SensorManager
                transQuat.w = gyro.attitude.x;
                transQuat.x = gyro.attitude.y;
                transQuat.y = gyro.attitude.z;
                transQuat.z = gyro.attitude.w;
                transQuat = Quaternion.Euler(90, 0, 0)*transQuat;//change axis around

         }
```

```
        if (DefaultOrientationIsPortrait){
               transform.Rotate(new Vector3(0, 0, 1), 90.0f); // apply z rotation to
                                                              // correct camera orientation
        }
}
```

Note that since the quaternion obtained from Unity is not in accordance with the output of the Android Sensor Manager, we have to re-order the elements in the 4-dimensional vector [W X Y Z] to conform standard quaternion definition. Meanwhile, in order to align the device orientation to the camera viewing angle in Unity 3D, we need to apply an additional 90-degree rotation in pitch direction.

The *DefaultOrientationIsPortrait* check in the code snippet handles the difference in the coordinate system for devices that have portrait as their default orientation (shown in Figure 2). If we don't apply the additional 90-degree rotation along the z-axis, we will get incorrect motion for the X and Y axis as shown in Figure 3 below.

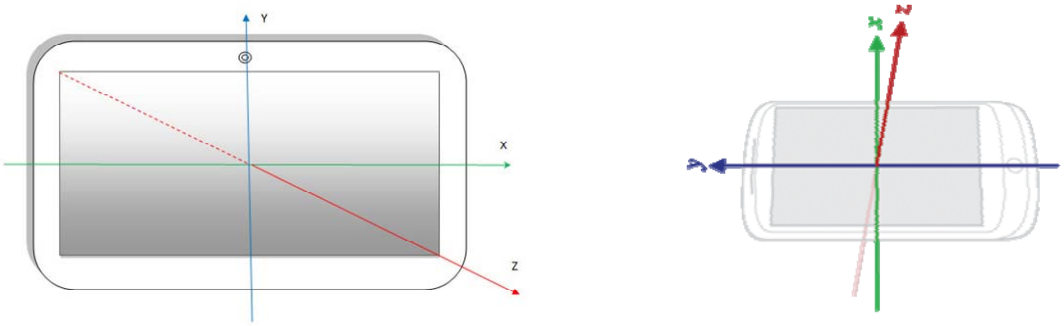


Figure 2. A device that has landscape as a default orientation (left) and another device that has portrait as default orientation (right).

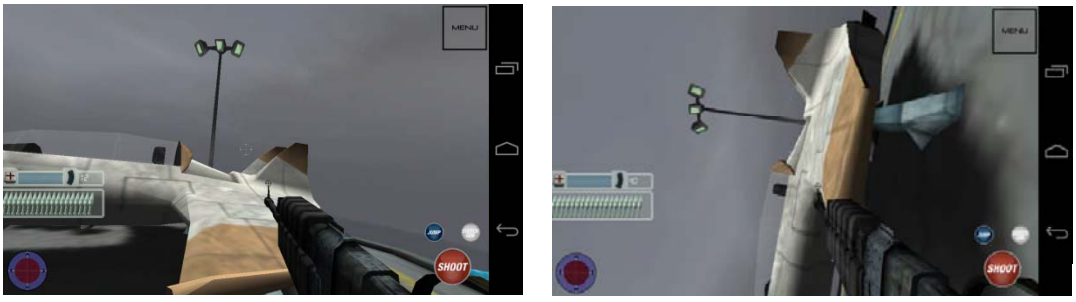


Figure 3. FPS game screen on a portrait device with (left) and without (right) the *DefaultOrientationIsPortrait* check. Notice the 90-degree rotation on the scene due to the coordinate system mismatch.

Android's Display class (http://developer.android.com/reference/android/view/Display.html) can be used to determine whether the natural orientation of the device is portrait or landscape. To get an instance of the Display class, use the function getWindow().getWindowManager().getDefaultDisplay():

```
Display default = getWindow().getWindowManager().getDefaultDisplay();

If( default.getRotation() ==Surface.Rotation_90 || default.getRotation() == Surface.Rotation_270)
{
        //Device's default orientation is portrait
        DefaultOrientationIsPortrait = true;
}
```

Note that Unity currently does not support Android's Display class API (current version is 3.4). The developer needs to create a Unity plug-in and include the above code snippet to expose the API to Unity applications. The plug-in is a .jar file that will be combined inside the Unity application that the developer creates. For more information on how to create a plug in for Android refer to the document at http://docs.unity3d.com/Documentation/Manual/PluginsForAndroid.html.

5.2 Advantages of Motion-Based User Interface

Compared to the conventional mouse or touch screen control, the Motion Interface control improves the gaming experience in many ways:

- Controlling the viewing angle with the mouse requires continuous integration and clamping of rotation parameters for the X and Y axis, which will eventually induct errors and lags in rotation (user needs to reposition the physical location of the mouse repeatedly). Motion interface (e.g. Rotation Vector) offers one-to-one matching between the device orientation and the game viewing angle, which gives a realistic feel to the gamers.
- Tweaking of the mouse sensitivity parameters needs an extra effort for smooth and optimized rotation which is not required, while using Rotation Vector is more convenient and precise.
- Device motion is a more natural way to navigate around the scene instead of using the mouse. Also you don't need to control the navigation of a game through two different devices i.e. keyboard and mouse.

6. References

[1] http://unity3d.com/support/documentation/Manual/Unity%20Basics.html
[2] http://en.wikipedia.org/wiki/Unity_%28game_engine%29
[3] http://developer.android.com/reference/android/hardware/SensorEvent.html
[4] http://en.wikipedia.org/wiki/Euler_angles
[5] http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation
[6] http://en.wikipedia.org/wiki/Rotation_matrix
[7] http://unity3d.com/support/documentation/Components/gui-Layout.html

APPENDIX

APPENDIX A: FIRST PERSON WALKING SCRIPT EXAMPLE

Scripting Basics

In Unity, scripting refers to building custom behaviors based on some logic or sequence of events and to associate it with a game object i.e. it could possibly be defined as an action associated with a game object. Different functions inside the script objects are called upon when a certain event occurs. The most used ones being the following:

Update:
This function is called before rendering a frame. This is where most game behavior code goes, except physics code.

FixedUpdate:
This function is called once every physics time step. This is the place to do physics-based game behavior.

LateUpdate:
LateUpdate is called every frame, if the behavior is enabled.

Awake:
Awake is called when the script instance is being loaded.

Start:
Start is called just before any of the Update methods are called for the first time.

Code outside any function:
Code outside functions is run when the object is loaded. This can be used to initialize the state of the script.

Event Handlers:
You can also define event handlers. These all have names starting with On, (i.e. OnCollisionEnter). To see the full list of predefined events, see the documentation for MonoBehaviour.

First Person Shooter Example

To illustrate the scripting capabilities of the Unity 3D game engine, we will write a script that will control the walking and jumping of a character in the game.

Figure 4 illustrates the basic game flow of Unity First Person Shooter game:

- Arena is the game environment that should be created in some 3D modeling tool like Maya or 3D studio max and then can later be imported in Unity 3D as an asset. It is the place where users are supposed to navigate, explore, and kill the enemies.
- First Person Shooter character consists of a camera object that is supposed to navigate the whole scene. Wherever the user points the camera, it would fire or navigate there in the same

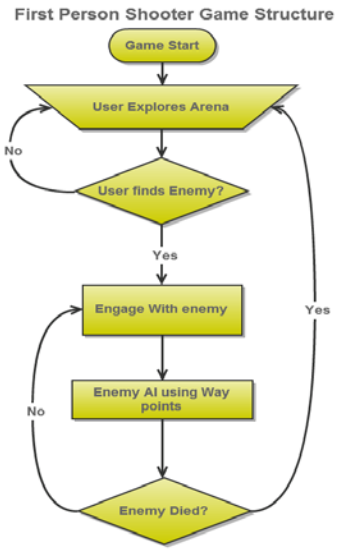


Figure 4. FPS Game Flow

direction, based on inputs.

- Enemy character is a robot, its movement is controlled using AI through waypoints logic to target and shoot the character. Waypoints are a point's graph in world space that actually define a possible path that one can take.

Our main goal in this section is to replace touch interface with Motion Interface, this requires us to first setup the FPS project.

To setup the FPS project in Unity 3D, please refer to the following links:

http://download.unity3d.com/support/resources/files/FPS_Tutorial_1.pdf
http://download.unity3d.com/support/resources/files/FPS_Tutorial_2.pdf
http://download.unity3d.com/support/resources/files/FPS_Tutorial_3.pdf

Before moving on to build the motion interface for FPS game, drag and drop the FPSMove script on the first person player to control the walking of player.

Unity3d game engine comes with a variety of ready to use components. For this demonstration, we will be using the Joystick component from the Unity3D standard mobile assets.

From Create drop-down menu of Project tab, select Javascript and name this script FPSMove.

Double click this script to edit in Mono-Develop IDE. Declare variables to control the walking speed, jumping speed, and gravity for the character controller. Declare some GUI related variables to have textures that will change for jumping up and down. Here is the resulting code snippet:

```
var speed:float = 5; // walking speed
var jumpSpeed = 8.0;// jumping speed
var gravity = 20.0;// gravity
var jumpTextureUp: Texture;
var jumpTextureDown: Texture;
private var currJumpTexture = jumpTextureUp;
private var movement = Vector3.zero; // tracking movement x,y,z
var initDir:Vector3; // initial direction
var moveJoystick : Joystick; //Joystick that will keep the track
private var character : CharacterController;  //UnityEngine.CharacterController class
```

Note that the variables which are not defined as private can be configured through Unity Editor.

Next is to initialize the variables that we will be using in the FPSMove script. It is always better to do the initialization code by overriding Start() as it is called just once before calling the Update() function
So below is the code snippet for the same.

```
function Start(){

    character = GetComponent( CharacterController );
    currJumpTexture = jumpTextureUp;

}
```

For rendering GUI interface elements, you will need to implement OnGUI(). OnGUI is called every frame and takes care of rendering any GUI elements in the scene. Rendering and handling the events on GUI elements like button, progress bar, notification text, etc. should be handled with OnGUI().

For example, for changing the texture of the jump button when the user presses it, you should override the OnGUI function as follows:

```
function OnGUI(){
	if(!currJumpTexture){
		Debug.LogError("Assign a Texture in the inspector.");
	}else{
		GUI.DrawTexture(Rect(Screen.width - 160, Screen.height - 180,50,50),
currJumpTexture);
	}
}
```

Debug class is provided by the unity engine so that you can debug your application by printing log messages for different debug levels, such as verbose, error, or warning. These messages can be shown on LogCat (Tool provided with Android SDK) once deployed on your platform.

GUI Class allows you to draw and render GUI elements and to perform related functions. For more information please consult the Unity Script reference on the Unity website.

The code that supports your script logic should be included within the Update() function. Update() function is called every frame of the game and here it should handle the walking and jumping of the character. Below is the code snippet for the First Person Shooter game.

```
	function Update () {

		#if UNITY_ANDROID
		if(Time.timeScale != 0){

			if (character.isGrounded) {
				// We are grounded, so recalculate movedirection directly from
	axes
				movement = new Vector3(moveJoystick.position.x, 0,
	moveJoystick.position.y);
				movement = transform.TransformDirection(movement);
				movement *= speed;
				var touch : Touch;
				for (touch in Input.touches) {
					if (Rect(Screen.width - 160,
	130,50,50).Contains(touch.position)) {
						movement.y = jumpSpeed;
				}
			}
			currJumpTexture = jumpTextureUp;
			}else{
				currJumpTexture = jumpTextureDown;
			}
			// Apply gravity
			movement.y -= gravity * Time.deltaTime;
			character.Move(movement * Time.deltaTime);
		}
	    #else
		if(Input.GetButton("Forward")){
			transform.position += transform.forward * speed * Time.deltaTime;
		}
		if(Input.GetButton("Backward")){
			transform.position += -transform.forward * speed * Time.deltaTime;
		}
		if(Input.GetButton("Left")){
			transform.position += -transform.right * speed * Time.deltaTime;
		}
		if(Input.GetButton("Right")){
```

```
                transform.position += transform.right * speed * Time.deltaTime;
            }
        #endif
    }
```

The above code snippet controls the movement of a player by retrieving the x and y component of the position of moveJoystick and then transforms this vector with respect to the character transform by calling transform.TransformDirection.

```
movement = new Vector3(moveJoystick.position.x, 0, moveJoystick.position.y);
movement = transform.TransformDirection(movement);
movement *= speed;
```

Input is the class that is provided by Unity to handle the variety of input devices, such as touchscreen, keyboard etc. For a touchscreen interface, by checking if the jump button is pressed, we can check if there is any touch point that lies inside the rectangular region of the jump button. If so then there should be a jump by multiplying the y movement with jumpSpeed, i.e.

```
for (touch in Input.touches) {
        if (Rect(Screen.width - 160, 130,50,50).Contains(touch.position)) {
                movement.y = jumpSpeed;
        }
}
currJumpTexture = jumpTextureUp;
}else{
        currJumpTexture = jumpTextureDown;
}
movement.y -= gravity * Time.deltaTime;
```

Finally after the calculation of movement it is time to apply the movement on CharacterController that we acquired in the start function, i.e.

```
character.Move(movement * Time.deltaTime);
```

APPENDIX B: MOTION SENSING & INTERFACE

Motion Interface is defined as the way to interact with devices and to perform particular tasks using motion sensors and related concepts. There are sensors and concepts regarding motion interface that you should know before developing any motion interface.

Three basic sensors regarding motion interface are

- **Accelerometer:** This measures the acceleration applied to the device (Ad). Conceptually, it does so by measuring forces applied to the sensor itself (Fs) using the relation:
  - **Ad = - ∑Fs / mass**

  In particular, the force of gravity is always influencing the measured acceleration:
  - **Ad = -g - ∑F / mass**

  For this reason, when the device is sitting on a table (and obviously not accelerating), the accelerometer reads a magnitude of g = 9.81 m/s^2

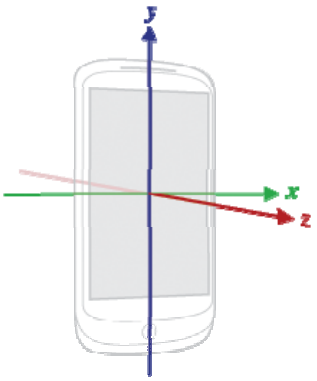


Figure 5: Coordinate System for Android Platform

- **Gyroscope:** This measures the rate of rotation around the device's local X, Y and Z axis. Rotation is positive in the counter-clockwise direction and negative in the clockwise direction.
- **Magnetometer / Magnetic Compass:** This measures the ambient magnetic field around the X, Y and Z axis. Note that the output of the gyroscope is not absolute, but is referenced to its initial condition. For example if you are measuring the absolute orientation of the device, you just can't do it using the gyroscope alone. You have to fuse compass data with gyroscope data to get the absolute orientation of the device.

  You should be familiar with the following mathematical concepts to use motion interface techniques effectively

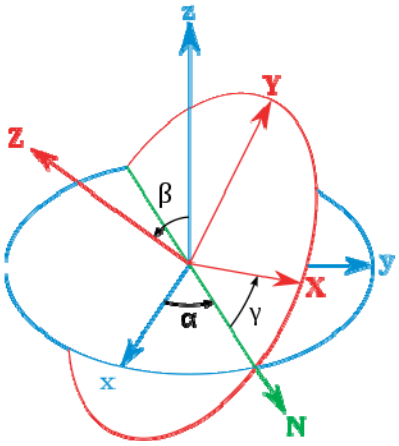


Figure 5: Euler Angles representation in 3D plane

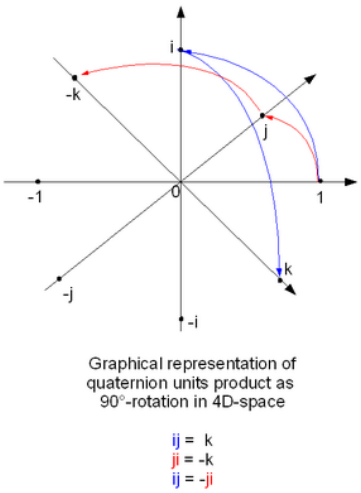
- **Euler angles**: Are a means of representing the spatial orientation of any frame (coordinate system) as a composition of rotations from a frame of reference (coordinate system). In the following, the fixed system is denoted in lower case (x,y,z) and the rotated system is denoted in upper case letters (X,Y,Z).

  α (or φ) is the angle between the x-axis and the line of nodes.
  β (or θ) is the angle between the z-axis and the Z-axis.

γ (or ψ) is the angle between the line of nodes and the X-axis.

In the figure-2 the xyz (fixed) system is shown in blue, the XYZ (rotated) system is shown in red. The line of nodes, labeled N, is shown in green.

- **Quaternion:** In pure mathematics language, quaternions are hyper complex numbers that are used to define the 3D rotation of a body. Any rotation in three-dimensions can be represented as an axis vector and an angle of rotation. Quaternions give a simple way to encode this axis-angle representation in four numbers and apply the corresponding rotation to position vectors representing points relative to the origin.



Graphical representation of quaternion units product as 90°-rotation in 4D-space

ij = k
ji = -k
ij = -ji

Figure 6: Quaternion Representation in Hyper complex plane

- **Rotation Matrix:** In linear algebra, a rotation matrix is a matrix that is used to perform a rotation in Euclidean space. For example the matrix

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

rotates points in the xy-Cartesian plane counterclockwise through an angle θ about the origin of the Cartesian coordinate system. To perform the rotation using a rotation matrix R, the position of each point must be represented by a column vector v, containing the coordinates of the point. A rotated vector is obtained by using the matrix multiplication Rv. Since matrix multiplication has no effect on the zero vector (i.e., on the coordinates of the origin), rotation matrices can only be used to describe rotations about the origin of the coordinate system.

APPENDIX C: GUI DESIGN IN UNITY 3D

Every application and game requires on-screen controls in the form of buttons, menu etc. to perform certain application related tasks like pausing the application, or changing application settings. The design of Unity GUI is simple and rich, which allows users to build nice and effective UI designs. For example, the following code will create and handle a button with no additional work in the editor or elsewhere:

```
// JavaScript
function OnGUI () {
  if (GUI.Button (Rect (10,10,150,100), "I am a button")) {
          print ("You clicked the button!");
  }
}
```

In the following example we will illustrate the GUI design capabilities of Unity by implementing an example menu
So start by overriding OnGUI(),

```
void OnGUI()
{
       GUILayout.BeginArea(new Rect((int)(Screen.width/4), (int)(Screen.height*.2),
       (int)(Screen.width/2), Screen.height));
        SensorMenu();
        GUILayout.EndArea();
 }
```

There are two different modes you can use to arrange and organize your GUIs: Fixed and Automatic. Fixed Layout makes sense to use when you have a pre-designed interface to work from. Automatic Layout makes sense to use when you don't know how many elements you need up front, or don't want to worry about hand-positioning each Control. For example, if you are creating a number of different buttons based on Save Game files, you don't know exactly how many buttons will be drawn. In this case Automatic Layout might make more sense. It is really dependent on the design of your game and how you want to present your interface. To use Automatic Layout, write GUILayout instead of GUI when calling control functions.

GUILayout.BeginArea()Begin a GUILayout block of GUI controls in a fixed screen area, and GUILayout.EndArea() ends the matching GUILayout.BeginArea();

```
void SensorMenu()
{
       GUILayout.BeginHorizontal();
       GUILayout.Label("Sensors: ", menuHeader, GUILayout.Height(Screen.height/10),
       GUILayout.Width(Screen.width/8));
        string text;
        if (camMove.gyroSet)
            text = "Gyro\nOn";
        else
            text = "Gyro\nOff";
       camMove.gyroSet = GUILayout.Toggle(camMove.gyroSet, text, selectionStyle,
                                          GUILayout.Height(Screen.height / 10),
                                    GUILayout.Width(Screen.width / 8));
        if (camMove.accelSet)
            text = "Accel\nOn";
        else
            text = "Accel\nOff";
        camMove.accelSet = GUILayout.Toggle(camMove.accelSet, text, selectionStyle,
                                                    GUILayout.Height(Screen.height / 10),
       GUILayout.Width(Screen.width / 8));
```

```
        if (camMove.compSet)
            text = "Mag\nOn";
        else
            text = "Mag\nOff";
        camMove.compSet = GUILayout.Toggle(camMove.compSet, text, selectionStyle,
                                                    GUILayout.Height(Screen.height / 10),
       GUILayout.Width(Screen.width / 8));
      GUILayout.EndHorizontal();
}
```

GUILayout.BeginHorizontal() and GUILayout.BeginVertical() will begin the area where the controls can be arranged in a horizontal or vertical fashion. Different styles for Control can be developed using GUIStyle class, here we are using two different styles i.e. selectionStyle and menuHeader style for labels and toggle buttons.

```
        selectionStyle = new GUIStyle();
        selectionStyle.normal.background = selectionFalse;
        selectionStyle.alignment = TextAnchor.MiddleCenter;
        selectionStyle.font = menuFont;
        selectionStyle.onNormal.background = selectionTrue;

        menuHeader = new GUIStyle();
        menuHeader.normal.background = menuHeaderBackground;
        menuHeader.alignment = TextAnchor.MiddleCenter;
        menuHeader.font = menuFont;

//where selectionFalse and selectionTrue are the instances of Texture2D class. You can
assign a
// texture to this from within Unity Editor to customize your control look.
```