UTS
UNIVERSITY OF TECHNOLOGY SYDNEY

ASSIGNMENT COVERSHEET

UTS: ENGINEERING & INFORMATION TECHNOLOGY

SUBJECT NUMBER & NAME	NAME OF STUDENT(s) (PRINT CLEARLY)	STUDENT ID(s)
Application Development in The iOS Environment 41889	Mark Macdonald Qi Hao Yin Padam Rao Karan Parikh *SURNAME* *FIRST NAME*	99130039 12595791 12497158 99180325

STUDENT EMAIL	STUDENT CONTACT NUMBER
99130039student.uts.edu.au 12595791student.uts.edu.au	0420 880 132 0423 578 495

NAME OF TUTOR	TUTORIAL GROUP	DUE DATE
Wei Wang	Group 8	08/06/2017 – 2pm

ASSESSMENT ITEM NUMBER & TITLE

Assessment Task 3

☐ I confirm that I have read, understood and followed the guidelines for assignment submission and presentation on page 2 of this cover sheet.
☐ I confirm that I have read, understood and followed the advice in the Subject Outline about assessment requirements.
☐ I understand that if this assignment is submitted after the due date it may incur a penalty for lateness unless I have previously had an extension of time approved and have attached the written confirmation of this extension.

Declaration of originality: The work contained in this assignment, other than that specifically attributed to another source, is that of the author(s) and has not been previously submitted for assessment. I understand that, should this declaration be found to be false, disciplinary action could be taken and penalties imposed in accordance with University policy and rules. In the statement below, I have indicated the extent to which I have collaborated with others, whom I have named.

Statement of collaboration:

[illegible] [illegible] [illegible] Padam

Signature of student(s) ______________ ______________ **Date** 07/06/2017

ASSIGNMENT RECEIPT

To be completed by the student if a receipt is required

SUBJECT NUMBER & NAME	NAME OF TUTOR

SIGNATURE OF TUTOR	RECEIVED DATE

STYLE GUIDE for ASSIGNMENT SUBMISSION

Before submitting an assignment, you should refer to the policies and guidelines set out in the following:

- FEIT Student Guide
- UTS Library - referencing
- HELPS - English and academic literacy support
- UTS GSU - coursework assessment policy and procedures

Unless your Subject Coordinator has indicated otherwise in the Subject Outline, you must follow the instructions below for submission of assignments in the Faculty of Engineering and Information Technology.

Writing style

It is usually best to write your initial draft in the default settings of your software without formatting. Use the following guides in your writing.

Purpose and audience: use the correct genre and language style expected for the particular task.

Language: use 'plain English' for all technical writing. More information about this language style can be found at www.plainenglish.co.uk/free-guides.html.

Use spelling and grammar software tools to check your writing. Edit your document.

Standards: always use:

- Australian spelling standards (Macquarie Dictionary)
- SI (International System of Units) units of measurement
- ISO (International Organisation for Standardisation) for writing dates and times for international documents. For example **yyyy-mm-dd** or **hh-mm-ss**. However, for most applications it is more helpful to present the date in full as **26 August 2016**.

Graphics and tables should:

- be numbered
- have an appropriate heading and/or caption
- be fully labelled
- be correctly referenced.

Presentation

Unless otherwise instructed, all assignment submissions should be **word processed** using spell-check and grammar-check software. Work should be well **edited** before submission. Use the following default settings:

Page setup: set margins at no less than 20mm all around.

Paper: print on A4 bond, double-spaced and preferably double-sided, left justified.

Font: use the software default style to provide consistency. The recommended style includes:

- 10-12 pt font
- consistent formatting with a limited number of fonts
- lines no more than 60 characters (use wider margins or columns if you need to make lines shorter)

Header should include:

- your name and student number
- the title of the paper or task.

Footer should include the page number and current date.

Cover sheet and statement of originality: all work submitted for assessment must be the original work of the student(s) submitting the work. A standard faculty cover sheet (see over) must be attached to the front of the submission. Any collaboration between the submitting student and others must be declared on the cover sheet.

Referencing

All sources of information used in the preparation of your submission must be acknowledged using the Harvard system of referencing. This includes all print, video, electronic sources.

Phrases, sentences or paragraphs taken verbatim from a source must be in quotation marks and the source(s) cited using both **in-text** referencing and a **reference list**.

Plagiarism is the failure to acknowledge sources of information. You should be fully aware of the meaning of plagiarism and its consequences both to your marks, position at the university and criminal liability. The plagiarism in your assignment submissions can be assessed both in hard copy and in soft copy through software such as Turnitin.

The UTS Library and UTS HELPS (web links above) provide extensive information for students on referencing correctly to support you in avoiding plagiarism.

Executive Summary

This report provides an in-depth analysis into the key features and functionality offered by the Apple developed framework, SpriteKit and CoreMotion. Sprite Kit is graphics rendering and animation API which assists in the development of 2D games, optimised for animation systems, physics simulation and event handling support. CoreMotion is another robust framework offered by Apple which provides motion and environmental data from the onboard hardware of a device, allowing motion control and detection to be implemented into an application.

The scope of this report is limited to the understanding of both SpriteKit and CoreMotion frameworks by; identifying their key features and functionality, description of these key features from both frameworks along with their practical implementation extracted from working code example and illustrating other practical use cases.

This reports starts with describing the frameworks. Followed by framework's key features, functionalities and the practical implementation of these features in form of code snippets. These code snippets have been obtained from working code example. The working code example is demonstration app, which is a 2D IOS game titled 'WestLand'. This game has been written with aim to utilize the key features of both frameworks. Following are capabilities of the game; create a player and shoot enemies; Move the player left to right on mobile screen through tilt gestures.

The key features that have been utilised for the game from SpriteKit are; SKView to provide scenes for animation and rendering, SKSpriteNode to draw gameplay elements, SKAction to implement in game events and SKTextureAtlas to store a collection of textures. For CoreMotion the application leverages the framework's CMMotionManager, which is used to obtain accelerometer data for the applications gameplay.

SpriteKit and CoreMotion are designed to be both intuitive and easy to use Frameworks within the Apple OS environment. They provide functionality that is battery efficient, customizable and fully compatible with all Apple hardware and operating systems.

SpriteKit

SpriteKit is a framework, developed by Apple, that is designed to help create powerful 2D games on all Apple operating systems, including; iOS, macOS, tvOS and watchOS. The framework allows developers to easily create high quality, battery efficient games that boast a wide range of functionality and depth, while also removing the need to download extra libraries or handle complex non-native API's such as OpenGL.

As a robust, feature filled framework, SpriteKit provides developers with a wide variety of functionality and customizability that aid in the creation of great 2D games that can utilize hardware effectively and minimise battery consumption. The core aspect of this framework is the graphics rendering capabilities that it provides, which runs a traditional rendering loop that alternates between determining the contents of and rendering frames (SpriteKit | Apple Developer Documentation, n.d.). This method then allows developers to determine the contents that frames display and how the content changes through different means. Included with the graphics rendering capabilities, the framework also offers a wide variety of features such as; animating, resizing, detecting collisions, applying gravity to or even generating new lighting effects for in-game items.

While boasting a great amount of functionality, the SpriteKit framework is primarily laid out as a hierarchical tree of structured nodes. This is done through a wide variety of class types provided by the framework, which range from the creating and displaying the gameplay elements of a game, to applying constraints and features to them, such as unit collision or gravity effects. These classes offered by the framework are the primary source of the usability and functionality that it can offer developers looking to make stellar 2D games.

Core Structure and Classes

While SpriteKit offers a range of classes and functionality to allow developers to maximise the full potential of their games, there are still many core classes that are used in development that are required to allow the framework to create, display and manage what is being displayed on the screen. The core classes used for this include SKView, SKScene and SKNode. These classes are used together to create the hierarchical tree of structured nodes (fig 1.0) that is core to the framework and allows the program to know where and what to display on the screen.

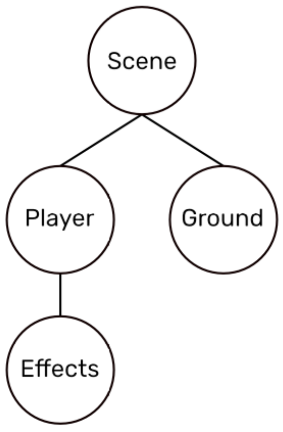


Figure 1

Each 2D game created using SpriteKit has multiple scenes that are used to split up content within the game. This can vary from different game level and stages, to menus and post-game screens. Created using the SKScene class, a scene holds sprites and other content that will be rendered on the screen, as well as per-frame logic and content processing that is executed automatically when the scene is present on the screen (SpriteKit | Apple Developer Documentation, n.d.). The rendering of these scenes are then handled by the SKView class, a subclass of UIView. SKView's display instances of an SKScene and can either be combined with multiple views for each game scene or can be used with only a single SKView that can switch between different scenes, utilising the SKTransition class to animate the transitions. The view class also provides some important functionality to gameplay, including the 'paused' Bool property which allows for a scene and gameplay to be easily paused and resumed.

Another fundamental class for the SpriteKit framework is the SKNode class, which has the primary role of providing baseline behaviour that other node classes can use (SKNode - SpriteKit | Apple Developer Documentation, n.d.). SKNodes, which SKScene is a subclass of, acts as the root node for a tree of node objects and applies its properties to its descendants. A node's actions are processed during every loop from a SKScene and its positioning is specified by the coordinate system of its parent, which also applies other properties to its content and the content of its descendants. An example of this is the rotation of a node will also rotate all descendants of that node. This allows developers to build a complex image using a tree of notes and apply various attributes and changes to the entire image by adjusting the topmost nodes properties (SpriteKit | Apple Developer Documentation, n.d.).

In the following sections we are discussing about key objects and classes for SpriteKit along with their code example description used in application for demonstration, "WestLand".

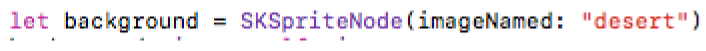
Drawing Content

SKSpriteNode

While the SKNode class doesn't perform any drawing of its own, it is a core aspect of the framework as it works with subclasses, such as SKSpriteNode, to display their content and allows for fast and effective changes to be applied to multiple items within a scene during a single frame. They assist in managing and maintaining the flow of displayed content of all nodes within a scene and are crucial to the functionality of the framework. SKSpriteNode which draw rectangular textures, images or colours and functions as the basic building block to the majority of a scene's content (SKSpriteNode - SpriteKit | Apple Developer Documentation, n.d.)

The following code snippet demonstrates creation of background node. This node is being initialised with image named "desert" that provides background to game.

Code Snippet 1.1

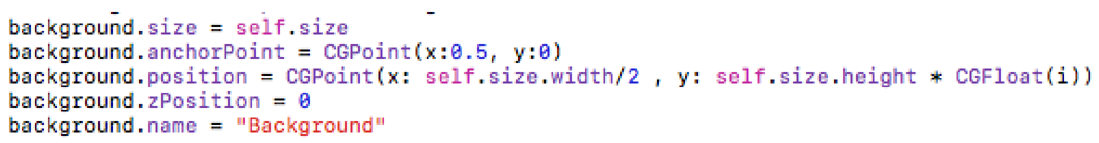


```
let background = SKSpriteNode(imageNamed: "desert")
```

When it comes to nodes that can display visual content such as shapes, images and videos the SpriteKit framework offers a wide variety that can be combined to create visually stellar games. This can include basic classes, such as SKLabelNode, which assists in displaying text labels, to more extensive classes, such as SKSpriteNode which is a crucial aspect of the SpriteKit framework and allows developers to display custom images and shapes, while also applying custom shaders, colours, lighting and rendering effects to them. They contain a vast amount of properties, such as the image, colour and size of the sprite, as well as its anchor, shaders and more. Combined with this, and their easy management through the SKNode class, SKSpriteNode's are a flexible and core class to this framework and provide an incredible amount of customizability that developers can use to create high quality, visually impressive games.

The following code snippet illustrates flexibility and customizability capabilities of SKSpriteNode. It displays the use of different properties and implementation of displays on the background node created in code snippet 1.1
The properties set on background node are, size property that matches the background node to size of frame, anchorPoint defines the point in the sprite that correlate with the nodes position. Using position property, position of the background can be horizontally centred and vertically made to change according to (i) value.

Code Snippet 1.2

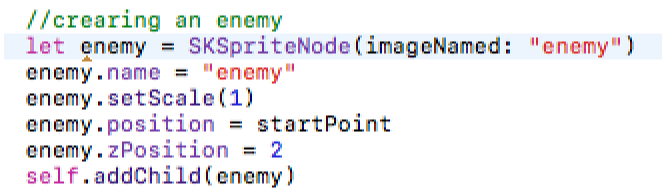


```
background.size = self.size
background.anchorPoint = CGPoint(x:0.5, y:0)
background.position = CGPoint(x: self.size.width/2 , y: self.size.height * CGFloat(i))
background.zPosition = 0
background.name = "Background"
```

SKTexture, SKTextureAtlas

The representation of images can be rendered by SKTexture objects. It provides the capability to apply same image to multiple sprites. SpriteKit can manage large number of images without compromising the performance by simply loading the texture explicitly.
In below code snippet 1.3 using SKSpriteNode, we are creating an enemy player. **imageNamed** creates a textured sprite node. Textured sprites are primary way to show custom artworks on view. In this case it represents an enemy player. We can also set some properties like the scale, position(CGPoint) and zPosition of the node.

Code Snippet 1.3

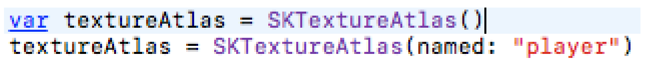


```
//crearing an enemy
let enemy = SKSpriteNode(imageNamed: "enemy")
enemy.name = "enemy"
enemy.setScale(1)
enemy.position = startPoint
enemy.zPosition = 2
self.addChild(enemy)
```

By creating a sprite this way, SKtexture object is created and attached to the node. It loads the texture data automatically. We don't need to worry about memory management here. SpriteKit automatically delete the texture data whenever sprite is removed from the screen or no longer visible.

Further, related textures can be grouped by SKTextureAtlas, which will be used to animate player and render background of level without compromising performance.

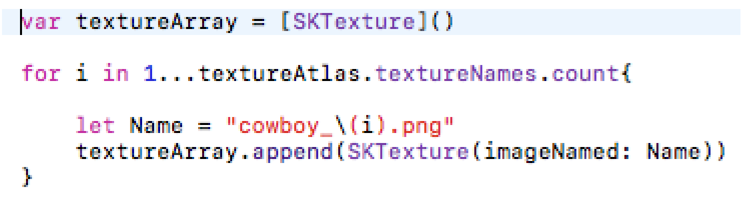
Code Snippet 1.4



```
var textureAtlas = SKTextureAtlas()
textureAtlas = SKTextureAtlas(named: "player")
```

In above code snippet we are initializing SKTextureAtlas() to textureAtlas. In second line we are pointing to the folder player which contains collection of sprites.
Afterwards using textureNames.count we can loop through all the sprites in that folder. We are storing all this sprites to SKTexture array.

Code Snippet 1.5



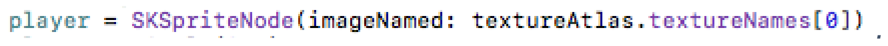
```
var textureArray = [SKTexture]()

for i in 1...textureAtlas.textureNames.count{

     let Name = "cowboy_\(i).png"
     textureArray.append(SKTexture(imageNamed: Name))
}
```

Code snippet 1.6 displays the level of ease for animating the player.

Code Snippet 1.6



```
player = SKSpriteNode(imageNamed: textureAtlas.textureNames[0])
```

Apart from the core drawing node, there are multiple classes offered by the SpriteKit framework that allows it to assist in creating high quality gameplay visuals and interactivity. This is possible via SKVideoNode. The SKVideoNode provides developers ability to display video content within their games, which can be used to display custom animation and outline visual behaviours in the game. This includes gameplay cut scenes to enhance a game's story, as well as visual effects such as a 'burst' when a bubble is popped.

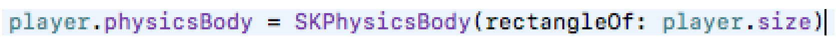
Simulating Physics

SKPhysicsbody

One of the largest features that the SpriteKit framework offers to developers is an inbuilt physics simulation that can be implemented into scenes and onto nodes. Through a variety of SpriteKit classes, developers are able to respond to collision and contact events, as well apply forces such as gravity and magnetism to nodes within a scene. When a node has physics elements applied to it, it uses the position and orientation of the node to place itself in the simulation, as well utilize innate properties, such as its velocity, mass and density, to define how it moves and how the applied physics forces should perform in the simulation (SpriteKit | Apple Developer Documentation, n.d.).

One of the core classes to this feature includes the SKPhysicsBody class. This adds physics simulations to a node and is called exactly when a new frame is processed by the scene. It handles the physics calculations that will be applied to the node and updates itself with new positions and orientations (SKPhysicsBody - SpriteKit | Apple Developer Documentation, 2017). These calculations include gravity, friction, contact and collisions with other bodies. In our 'WestLand' example we are using SKPhysicsContactDelegate to detect when collision of two physics bodies with each other. To implement this first give a physics body using physicsBody property to nodes, in our case to player, enemy and bullet (code snippet 1.7).

Code Snippet 1.7

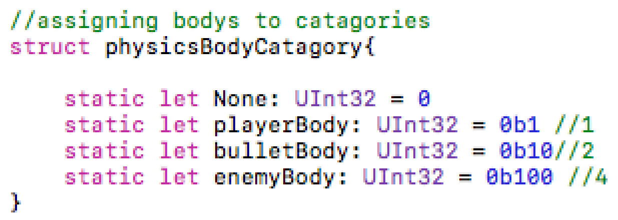


```
player.physicsBody = SKPhysicsBody(rectangleOf: player.size)
```

Here we are using rectangle shaped body for the player node. There are other options like circular, polygonal, alpha channel. Different shapes can affect the performance of the body. Circular body offers the best performance and can be significantly faster than other physics bodies (SKPhysicsBody - SpriteKit | Apple Developer Documentation, 2017).

Furthermore, to use physics we have to also use different categories for objects that appears on the screen.
This can be illustrated by following code snippet 1.8.

Code Snippet 1.8

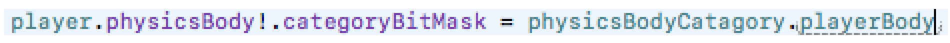


```
//assigning bodys to catagories
struct physicsBodyCatagory{

     static let None: UInt32 = 0
     static let playerBody: UInt32 = 0b1 //1
     static let bulletBody: UInt32 = 0b10//2
     static let enemyBody: UInt32 = 0b100 //4
}
```

In the above code snippet there are four categories to identify the contact between bodies easily. Here we have created different bitmap constants as a category for objects that will act as unique identifiers.
Next step is to assign the category to node. This can be implemented as below.

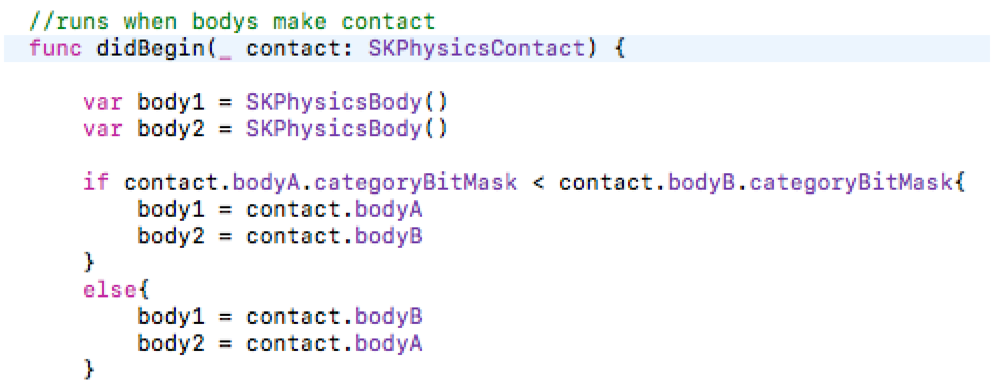
Code Snippet 1.9



```
player.physicsBody!.categoryBitMask = physicsBodyCatagory.playerBody
```

The next step requires providing definition to function didBegin(_ contact: SKPhysicsContact) that identifies the contact between two objects and assign the object to body A or body B based on lower bitMask value. The reason for doing this is that our pre-defined bit mass assigned to object can always stay on numerical order thus enabling us in identifying the bodies that made contact.

Code Snippet 1.10



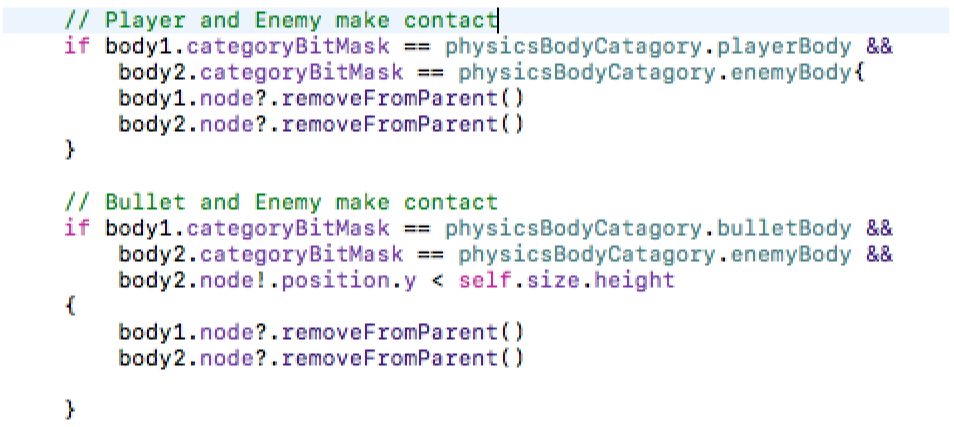
```
//runs when bodys make contact
func didBegin(_ contact: SKPhysicsContact) {

    var body1 = SKPhysicsBody()
    var body2 = SKPhysicsBody()

    if contact.bodyA.categoryBitMask < contact.bodyB.categoryBitMask{
        body1 = contact.bodyA
        body2 = contact.bodyB
    }
    else{
        body1 = contact.bodyB
        body2 = contact.bodyA
    }
```

In ‘WestLand’ either enemy is killed with contact by bullet or player when in contact with enemy. For this we define the following conditions so that we can kill either player or enemy.

Code Snippet 1.11



```
    // Player and Enemy make contact
    if body1.categoryBitMask == physicsBodyCatagory.playerBody &&
        body2.categoryBitMask == physicsBodyCatagory.enemyBody{
        body1.node?.removeFromParent()
        body2.node?.removeFromParent()
    }

    // Bullet and Enemy make contact
    if body1.categoryBitMask == physicsBodyCatagory.bulletBody &&
        body2.categoryBitMask == physicsBodyCatagory.enemyBody &&
        body2.node!.position.y < self.size.height
    {
        body1.node?.removeFromParent()
        body2.node?.removeFromParent()

    }
```

This was the demo for physics body can do. Through the implementation of this classes, as well as others offered by the SpriteKit framework, developers are provided the ability to create immersive games that can correlate to real life physics, allowing for the easy development of a broader range of games and applications.

Lighting, Actions and Other Features

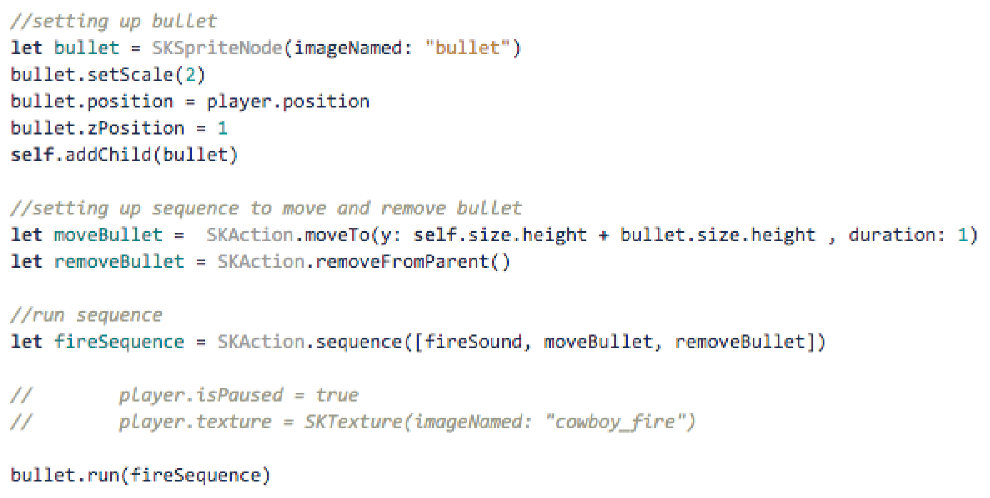
The SpriteKit framework is a highly robust framework that offers developers a wide variety of functionality that assists in the creation of high quality, battery efficient games. The framework offers classes that assist in the management of nodes through constraints, classes which provide the ability to play audio sounds that can also be 3D spatially aware from within a games scene, as well as classes that allow developers to build a detailed scene with tiles that lower memory overhead. The framework provides developers with all the required tools to create high quality 2D games.

Another strong feature that the framework provides is the ability to add custom lighting to a scene. Managed through the SKLightNode class, they are invisible nodes that can perform actions within a scene and provide lighting interactions with other sprites that are set to interact with them. This allows custom lighting effects to be implemented into a game, which allows for more in depth and realistics games that implement real life aspects. It can be used by developers in a multitude of ways, including adding a sun to their game, and allows them avoid complex API’s that would otherwise be needed to resolve the lighting calculations.

Another noteworthy feature of the framework is the SKAction class. This class represents an action that is executed by a node within a scene (SKAction - SpriteKit | Apple Developer Documentation, n.d.). An important aspect to the framework, actions can change the structure and content of a node or scene and provide core functionality to a developer's gameplay. It provides the ability to create action events that can occurs by the player's input or by gameplay events that are predefined and dramatically increases the reactiveness and interactivity that developers can implement into their games and gameplay.

The code snippet 1.12 defines action of bullet using SKAction. Initially sprite named bullet has been initialised with image. Further its properties regarding position and scale have been set. A collection of actions to be executed sequentially can also be set with help of sequence function. Thus sound of bullet, followed by bullet’s movement and finally removal of bullet node from parent that is deleting it from scene have been accomplished passing the fire sequence to run method.

Code Snippet 1.12



```
//setting up bullet
let bullet = SKSpriteNode(imageNamed: "bullet")
bullet.setScale(2)
bullet.position = player.position
bullet.zPosition = 1
self.addChild(bullet)

//setting up sequence to move and remove bullet
let moveBullet =  SKAction.moveTo(y: self.size.height + bullet.size.height , duration: 1)
let removeBullet = SKAction.removeFromParent()

//run sequence
let fireSequence = SKAction.sequence([fireSound, moveBullet, removeBullet])

//        player.isPaused = true
//        player.texture = SKTexture(imageNamed: "cowboy_fire")

bullet.run(fireSequence)
```

Through the robust and feature filled framework that is SpriteKit, Apple have created a high powered and efficient framework that provides developers a simple avenue to create amazingly powerful 2D games. With the implementation of the discussed features and functionality offered by the framework, developers can utilized it to create visually stellar, battery efficient, interactive applications that work perfectly for the development and implementation of 2D games and gameplay, but can also be expanded into other visually heavy and high functioning applications.

Practical Use Cases

SpriteKit Use Case #1 - Lighting	
Use Case Description	A developer is creating a 2D game using the iOS native SpriteKit framework. The app is a simple side scrolling game that incorporates day and night periods into its levels that work into its gameplay. The developer is looking for a simple method to add a 'sun' sprite on the screen that will display lighting effects to items being rendered on the screen, based on its position.
Required Functionality	Dynamic lighting effects that change based on the lights position.
Proposed Solution	To resolve this issue the developer could make use of the SpriteKit framework and its native lighting functions. This is offered in the class 'SKLightNode' and can be used to provide the desired lighting effects for the developer, as well as be repositionable to allow the developer to replicate a moving sun.
Example Solution (with code)	Creating a sprite that utilises SKLightNode and is displayed with a sun image (Loew, n.d.). 1. Create a sprite that will represent the 'sun' and assign its position, scale and image. 2. Then create and attach an SKLightNode to the sprite (Loew, n.d.).

	var light = SKLightNode();
	light.position = CGPointMake(0,0);
	light.falloff = 1;
	light.ambientColor = UIColor.darkGrayColor();
	light.lightColor = UIColor.whiteColor();
	_sunSprite?.addChild(light);
	3. Once complete and all properties are set, animate the sprite to move across the top of the screen.

Solution Discussion	The above code demonstrates the ease of use of the SpriteKit framework and how to implement an SKLightNode onto a sprite. Adjustable properties to the above include; lightColor which is the light it emits, ambientColor which is the background lighting it shines on all pixels, and falloff which dictates how far the light is visible. Through these properties and the implementation of a SKLightNode the developer can easily implement and manage a 'sun' within their application.

CoreMotion

CoreMotion is a framework which provides developers with the essentials to monitor motion data within any iOS and watchOS operating hardware. Furthermore, developers can apply the motion data to further enhance the user's experiences on applications that utilise motion, for instance expanding the possibilities of current navigation and game mechanics for mobile games.

CoreMotion provides more depth and functionality for users as it challenges the boundaries of common user interactions with devices such as tapping and swiping. With the CoreMotion framework, it pushes the capabilities of contemporary hardware and software. The core concept behind this framework is to extract and process raw data from sensors including the accelerometer, gyroscope and magnetometer, which are undoubtedly integrated within any current iOS devices. From this data, developers can determine the orientation of a device, whether it is portrait or landscape as well as detecting motion of the device for instance, the movement of the device on the X, Y and Z axis plane.

Core Structure and Classes

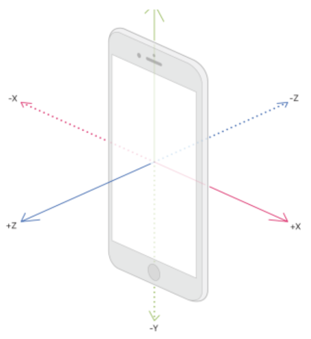
There are numerous classes which provide developers with information on how a device is handled, further enhancing their applications by offering the additional functionality. There are many detailed classes within the CoreMotion framework but they all revolve mainly around the CMMotionManager class. This class is necessary to monitor and manage the motion data in order for it become viable for the program to process the data.

CMMotionManager essentially provides developers with the ability to access the motion data from iOS devices. The motion data is categorised into four types; accelerometer data, rotation-rate data, magnetometer data, and other device-motion data. Additionally, once an instance of the CMMotionManager is created, developers are able to extract each of the data in its raw form (CMMotionManager | Apple Developer Documentation, n.d.). The primary step of accessing the data is ensuring that the appropriate sensor is integrated into the hardware system, within the CMMotionManager class, the program will confirm whether the feature is available. This class consist of two methods of sampling data, the first adds an operational queue which updates during a specified interval providing more reliable and accurate data. The other method includes sampling over a period of time, only showing the latest, updated data for more suitable for games as there is less overhaul. This provides flexibility for the developer to appropriately decide on which method is more suitable to create an efficient and robust game.

Accelerometer

As mentioned above, one of the more frequently used hardware features associated to the CoreMotion framework is the accelerometer. The accelerometer is a device which measures the force of acceleration, in the case of iOS development, the accelerometer measures the changes in velocity along a single axis (Getting Raw Accelerometer Events | Apple Developer Documentation, n.d.). However, this is impractical as developers need to grab data input from the X, Y and Z axis of the device, because of this Apple made it mandatory to install a three-axis accelerometer into all their iOS hardwares (fig 2.0). Through the use of the accelerometer, developers will have the ability to create more interesting applications via motion input.

Figure 2



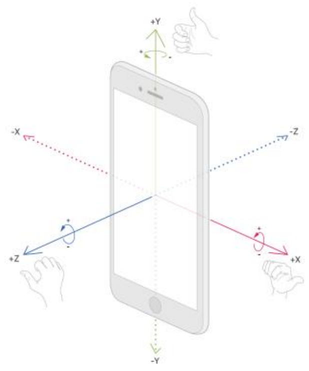


Figure 3

Gyroscope

Similarly to the accelerometer, the gyroscope is another fundamental feature that monitors the orientation of the iOS device. Whereas the the accelerometer detects a change in speed along the three axis, the gyroscope measures the rotation of the device across all three axis (fig 3.0). Developers can utilise the gyroscope to main the orientation of the device to create new methods of user interactions.

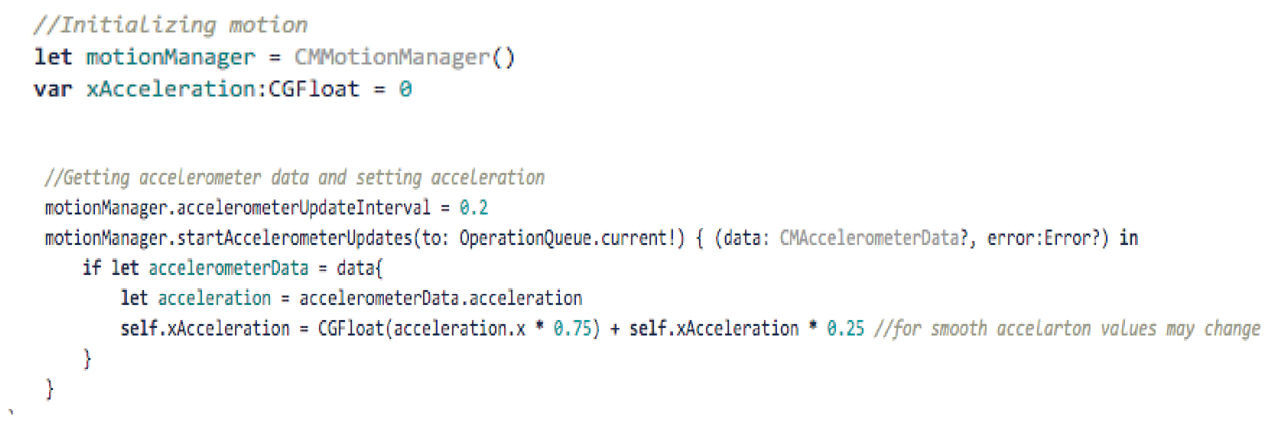
Although both accelerometers and gyroscopes are similar features, the characteristics of these two sensors provide developers with different sets of motion data. On its own, each of these features already give developers a wider range of functionality. Once a developer incorporates all these sensors into their projects, they will able to produce powerful and efficient applications with highly accurate information.

Despite an increase in the usage of CoreMotion in iOS development, the fact remains where other user interactions such as tap and swipe remain as favourites. As a result of this, the CoreMotion framework is under used and is only applied to a small genre of applications. However, this incredible and user friendly framework, CoreMotion allows developers to produce new and innovative motion based applications. Using the the CMMotionManager class to process information from the mentioned features above, developers can nurture this framework to its potential

Code Example

Initially object named motionManager is created for starting and managing motion services. Further using the motionManager object is used to get and set accelerometer data and setting acceleration code snippet 2.1.

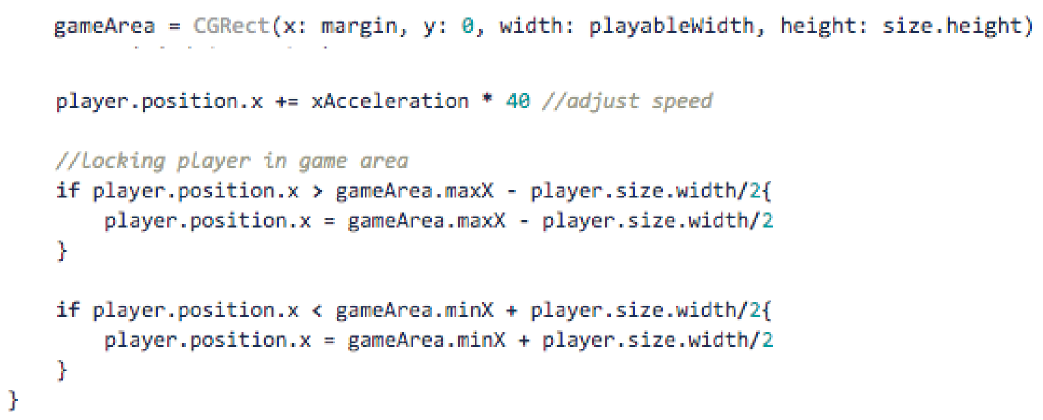
Code Snippet 2.1



```
//Initializing motion
let motionManager = CMMotionManager()
var xAcceleration:CGFloat = 0


 //Getting accelerometer data and setting acceleration
 motionManager.accelerometerUpdateInterval = 0.2
 motionManager.startAccelerometerUpdates(to: OperationQueue.current!) { (data: CMAccelerometerData?, error:Error?) in
     if let accelerometerData = data{
         let acceleration = accelerometerData.acceleration
         self.xAcceleration = CGFloat(acceleration.x * 0.75) + self.xAcceleration * 0.25 //for smooth accelarton values may change
     }
 }
```

Code Snippet 2.2



```
    gameArea = CGRect(x: margin, y: 0, width: playableWidth, height: size.height)

    player.position.x += xAcceleration * 40 //adjust speed

    //Locking player in game area
    if player.position.x > gameArea.maxX - player.size.width/2{
        player.position.x = gameArea.maxX - player.size.width/2
    }

    if player.position.x < gameArea.minX + player.size.width/2{
        player.position.x = gameArea.minX + player.size.width/2
    }
}
```

In code snippet 2.2 the player position is compared against game area and player size width. If the player size is more than half outside the screen. It would be push inside the screen with similar coordinates. This has been used to restrict the player from going out of screen when the screen is tilted completely left or right.

Practical Use Cases

CoreMotion Use Case #1 - Accelerometer	
Use Case Description	A student with a task of creating a simple game on an iOS platform device using the CoreMotion framework and in the Swift language. The student has decided to create a game where a player navigates an object to the end goal by tilting the hardware device.
Required Functionality	Moving an object based on the angle of device.
Proposed Solution	In order for the student to successfully capture the data from the data, the student needs to use the class 'CMMotionManager' to access the data from the iOS device. Then in order for the player model to move depending on the device, the player can input the data into the class SKPhysicsWorld to change the gravity of the scene, hence moving the player towards the desired direction.

Example Solution (with code)	Affect the gravity so that the player model follows how the device is being handled (Davidson).
	
	`let manager = CMMotionManager()`
	
	`manager.startAccelerometerUpdates()`
	`manager.accelerometerUpdateInterval = 0.1`
	`manager.startAccelerometerUpdatesToQueue(NSOperationQueue.mainQueue()) {`
	`    (data, error) in`
	`    self.physicsWorld.gravity = CGVectorMake(CGFloat(data?.acceleration.x) * 20, CGFloat(CGFloat(data?.acceleration.y) * 20)`
	`}`

Solution Discussion	The code begins with recording data from the accelerometer and setting the interval time of each sample at a one tenth of a second. Then an operation queue is created so that the data gathered from the accelerometer is used to change the gravity of the program according to the direction of the tilted device.

References

Allie, D. 2016, *SpriteKit From Scratch: Fundamentals*, Code Envato Tuts+. viewed 3 June 2017, <https://code.tutsplus.com/tutorials/spritekit-from-scratch-fundamentals--cms-26326>.

Bhatia, R. 2016, "Smartphone Sensors : Gyroscope, Accelerometer And Magnetometer : Basic Difference". *SmartFoneArena*. viewed 7 June 2017, <http://smartfonearena.com/smartphone-sensors-gyroscope-accelerometer-magnetometer-basic-difference/>.

CMMotionManager - CoreMotion | Apple Developer Documentation n.d., Developer.apple.com. viewed 5 June 2017, <https://developer.apple.com/documentation/coremotion/cmmotionmanager>.

Cook, N. 2014, *CmDeviceMotion*, NSHipster. viewed 5 June 2017, <http://nshipster.com/cmdevicemotion/>.

CoreMotion | Apple Developer Documentation n.d., Developer.apple.com. viewed 5 June 2017, <https://developer.apple.com/documentation/coremotion>.

Davidson, J. 2016, *Make a Maze Game! (Core Motion : Swift 2 in Xcode)*. [Online Video]. 3 February 2016. Available from: <https://www.youtube.com/watch?v=_moMI9nzbMU>. [Accessed: 7 June 2017].

Getting Raw Accelerometer Events - CoreMotion | Apple Developer Documentation n.d., Developer.apple.com. viewed 7 June 2017, <https://developer.apple.com/documentation/coremotion/getting_raw_accelerometer_events>.

Getting Raw Gyroscope Events - CoreMotion | Apple Developer Documentation n.d., Developer.apple.com. viewed 7 June 2017, <https://developer.apple.com/documentation/coremotion/getting_raw_gyroscope_events>.

Loew, A. n.d., *SpriteKit dynamic light tutorial*, Codeandweb.com. viewed 7 June 2017, <https://www.codeandweb.com/spriteilluminator/tutorials/spritekit-dynamic-light-tutorial>.

SKAction - SpriteKit | Apple Developer Documentation n.d., Developer.apple.com. viewed 5 June 2017, <https://developer.apple.com/reference/spritekit/skaction>.

SKNode - SpriteKit | Apple Developer Documentation n.d., Developer.apple.com. viewed 4 June 2017, <https://developer.apple.com/reference/spritekit/sknode>.

SKPhysicsBody - SpriteKit | Apple Developer Documentation 2017, Developer.apple.com. viewed 5 June 2017, <https://developer.apple.com/reference/spritekit/skphysicsbody>.

SKSpriteNode - SpriteKit | Apple Developer Documentation n.d., Developer.apple.com. viewed 5 June 2017, <https://developer.apple.com/reference/spritekit/skspritenode>.

SpriteKit | Apple Developer Documentation n.d., Developer.apple.com. viewed 4 June 2017, <https://developer.apple.com/reference/spritekit/>.