Juuso Koskenseppä

iOS game development

Mobile game development with Swift programming language and SceneKit framework

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Media Technology

Thesis

Date 13.11.2016

Author(s) Title Number of Pages Date	Juuso Koskenseppä Mobile game development with Swift programming language and SceneKit framework 44 pages 13.11.2016
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructor(s)	Kari Salo, Principal Lecturer

The purpose of the thesis was to create an iOS game that could be deemed complete enough, so it could be published in Apple's App Store. This meant fulfilling different guidelines specified by Apple.

The project was carried out by using Apple's new Swift programming language and SceneKit framework, with an intention to see how they work for iOS game development.

The immaturity of Swift programming language led to several code rewrites, every time a newer Swift version was released. There could have also been a reason to consider using protocol oriented programming instead of object oriented programming, to make the code simpler. The results of an user testing that was performed for the project revealed that it still needs better instruction before it can be considered to fulfil all Apple's guidelines.

The thesis showed that Swift programming language and SceneKit framework could be used to program an iOS game that fulfils Apple's guidelines without a big challenge. Also no compromises with the game design were required because of them.

Keywords	iOS, game development, Swift, SceneKit

Tekijä Otsikko Sivumäärä Aika	Juuso Koskenseppä Mobiilipelin kehitys Swift-ohjelmointikielellä ja SceneKit-sovelluskehyksen avulla 44 sivua 13.11.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaaja	Yliopettaja Kari Salo

Insinöörityön tarkoituksena oli luoda iOS-peli, jota pystyttäisiin pitämään tarpeeksi valmiina, että se voitaisiin julkaista Applen App Storessa. Tämä tarkoitti erilaisten Applen määrittämien ohjeistuksien täyttämistä.

Työ toteutettiin käyttäen Applen uutta Swift-ohjelmointikieltä ja SceneKit-sovelluskehystä, ja siinä tutustuttiin niiden toimintaan iOS-pelinkehityksessä.

Swift-ohjelmointikielen epäkypsyys johti useisiin koodin uudelleenkirjoituksiin aina, kun kielestä julkaistiin uusia versioita. Työssä olisi myös voinut olla syytä harkita protokolla-ohjelmoinnin käyttöä olio-ohjelmoinnin sijaan, jolloin koodista olisi voinut saada vielä yksinkertaisempaa. Pelille tehty käyttäjätestaus paljasti, että se kaipaa vielä parempia ohjeistuksia, ennen kuin sen voidaan todeta täyttävän kaikki Applen ohjeistukset.

Insinöörityö osoitti, että Swift-ohjelmointikieltä ja SceneKit-sovelluskehystä voidaan käyttää ilman suuria haasteita Applen ohjeistuksien mukaisen iOS-pelin ohjelmointiin. Niiden takia ei tarvinnut tehdä kompromisseja peliä suunniteltaessa.

Avainsanat	iOS, pelinkehitys, Swift, SceneKit

Contents

1 Introduction

The ways in which iOS games can be developed have evolved a lot recently. First with Apple releasing SpriteKit and SceneKit 2D-and 3D game frameworks with iOS 7 to make it easier to develop games, and now most recently with the release of Swift-language to make writing applications easier.

This thesis studies how to use SpriteKits, perhaps a lesser known and used sibling, SceneKit framework in combination with the new Swift language for game development. This is based on a game project that I started and completed by using the said technologies. The goal for this game project was to develop a full-fledged iOS game that is publishable to Apples App Store.

iOS games can currently be developed with multiple different frameworks and with a couple of different programming languages. The first chapter of this thesis answers why SceneKit and Swift were chosen as the graphics framework and programming languages for the game projects. The chapter also tells about the games actual design and how it was tested.

The following chapters show, with examples taken from the project, how the game was eventually programmed. This involved the usage of most of SceneKits features and thus the thesis can be used to learn some basics of how to use the framework with Swift. The chapter four lists actual testing results that were needed for perceiving if the projects objectives were reached.

The game has also been refactored a couple of times: first from Swift 1.2, which was the current version back when the project was started, to Swift 2.0 and then most recently to version 3.0. This constant refactoring was a major inconvenience that the final chapters cover, along with reflections that the project generated.

2 Project background

2.1 Project goals

The main goal of the project was to create a game that is mature enough that it can be published in Apples App Store. This goal demands that the application should be complete enough to get through Apple's application review process. To get through the review process the game should follow all the guidelines and requirements listed in App store review guidelines. The listed guidelines include some highly relevant parts for the project that are either clearly visible or indirectly visible to the user. (App Store Review Guidelines, 2016.)

A directly visible guideline for the potential user is iOS Human interface Guidelines that essentially tell how an application should look and feel when used. This means that the projects interface should be designed according to these guides. For example some important parts in the guide tell that any graphics and buttons should be clear on what they do, the application should use right terminology and give control to the user in choices. (iOS Human Interface Guidelines, 2016.)

Some guidelines that might not be as directly visible, but the project should follow, are App Programming Guide and iOS Data Storage Guidelines that ensure that the project works well and any possible data persistence is done in a right way (App Store Review Guidelines, 2016). The former entails that the application should run efficiently and use life cycle correctly (App Programming Guide for iOS, 2016). The guide also has several strategies of how to run specific tasks like state preservation properly (App Programming Guide for iOS, 2016). The latter guide tells what data should be saved and where (iOS Data Storage Guidelines, 2016).

The project also should not fall to the list of common rejection causes such as being buggy and crash prone, nor should it include any placeholder content (Common App Rejections, 2016). This meant that the project should be finished in its content and its delivery.

Some of the first warnings introduced in the App store review guidelines state "If your app looks like it was cobbled together in a few days, or you're trying to get your first practice app into the store to impress your friends, please brace yourself for rejection.

We have lots of serious developers who don't want their quality apps to be surrounded by amateur hour” (App Store Review Guidelines, 2016.). This means that even though the project was done independently, the project was also taken highly seriously and perfection was aimed for in everything.

2.2 Game design

The plan for the game design started from an idea where I wanted to do a game where player floats in space while trying to finish some kind of a mission. I also wanted to differentiate from the crowd and make the game in 3D instead of the easier 2D. For this purpose I deemed only first-person and third-person perspectives as a working alternatives on how the game is viewed. In the end I preferred first-person view as a better choice because it meant that I needed less graphics and because screen size was also limited. This is because the chosen perspective does not need a model for the player that would then constantly block part of the already small and easily crowded screen that smart phones have. This idea eventually split in to two different game modes: the main game and stripped down side game.

In the main game the player moves in space and, as was already decided, in first-person perspective. The mission for the player is to save the world from meteors, while also being chased by comets that try to destroy the player. Each round, an ever-increasing amount of meteors will spawn around the Earth and the player will be moved to starting position, with one comet spawning behind and in front of the player. The player has finite amount of ammunition to destroy the meteors before they hit the Earth that, however, can survive a total of five hits from meteors before the player loses the game. After a successful round a shopping screen is shown to the player, who can use it to purchase more ammunition, more movement speed, more armour to survive collisions with meteors and more health for Earth with credits that the player earns by destroying meteors. The comets that chase the player will increase with speed every round so player usually needs to at least purchase more speed and ammunition to survive further.

There are three kinds of meteors to destroy, with varying difficulty. The ratio of the more difficult to destroy ones to spawn against the easier ones increases with rounds. The easiest meteors to destroy will simply move towards Earth in straight line from their spawning position, while more difficult ones will do a little zigzag, making aiming at

them harder. The hardest meteors will move in straight line, but will be invulnerable for half a second for every second of being vulnerable. This almost always means that more than one ammunition is needed to destroy them successfully, because from long distances it is very difficult to estimate the time to shoot correctly.

As for the controls, the player is always moving forwards at set velocity and the player can turn by sweeping the phones screen. The player can also shoot projectiles that will move directly forwards from the player by tapping the screen. The map has invisible walls to limit the player's movement and a warning is displayed before the player collides with them: a collision means instant game over for the player. These walls are also used to remove any missed ammunition, to limit the amount of models and graphics in the game.

The simpler side game is basically a turret-game version from the main game. The player is now placed in the middle of the map where Earth was and the meteors will move towards the player at high velocity. The player can only turn by the Y-axis to look around, unlike in main game where the player can also rotate by the X-Z-axis to look up and down. The turret version has no shopping mechanics and the player has infinite amount of ammunition. The idea is just to try to survive as many rounds of the ever-increasing meteor onslaught as possible.

2.2.1 Game graphics

The game needed several custom graphics that were created either with Blender (3D models), Pixelmator (images) or Adobe Illustrator (logo) and most of these images required three different scales for different device screen sizes. There were multiple choices for applications that could have been used for these purposes as the personal requirements were if I had experience with the application and if I had access to it. The former requirement was usually the most decisive requirement as most graphics applications are quite expensive. For the 3D models Blender was the only free 3D graphics software I had former experience with so it was chosen for this task. For image manipulation I could have also used Adobe Photoshop, but as the image manipulation needs were really light in nature, Pixelmator was enough to deal with them and there was no need for the heavier Photoshop.

However, with Adobe Illustrator that was used for the vector graphics, the exportation to bitmap images was a bit tedious task, as was the task of creating multiple images of different scales with Pixelmator. These scaled images are needed by iOS so it can use image of the right size for different screen resolution. If I had to do this part of the project again, I would use Sketch for all the bitmap and vector images because the application can be used easily to export all the wanted scales of an image. This would have saved a lot of time when I was working on the projects graphics.

Image 1. Starting menu background image

Image 1 displays several of these created graphics. Showcased in the middle is the meteor model that is also used in the game that was created with Blender. Like the larger meteor, the smaller meteor fractures surrounding the image were similarly modelled in Blender. The logo was created with Illustrator and exported into different sizes with Pixelmator.

The icons for buttons were designed according to the iOS Human Interface Guidelines. The icons were kept simple and common recognizable patterns for the images were used such as “X” mark for closing. They were also used consistently throughout the application. (iOS Human Interface Guidelines, 2016.)

2.2.2 How the design was tested

A light testing was done throughout the developing phase to improve the design. These tests were done by letting people play the game with test device and then by asking how they would improve the game. This testing was not specifically planned nor was it documented because the acknowledged issues found in these tests were usually remedied right after the test. The intention of these small tests was to find out the most obvious issues and bugs before a real user testing for the game could be done.

A big issue that was out in these developing phase tests was how dark the game was. At first the games background was black to emulate space. This, however, meant that it was really hard for people to see anything from the screen when they were outside in a sunny weather. To alleviate this problem, the background colour was changed to this dark blue seen on image 1. This was mostly a compromise, because going for even lighter shades of colours meant that the illusion of flying in open space was mostly lost, the chosen colour, however, is on the edge that it is possible to see something even in sunny conditions but it still feels like floating in space.

Another issue found in these tests was the lack of an alert dialog when quitting the game. This was also deemed a big issue, because Apples iOS Human Interface Guidelines talk about how the application should not make the choices for the user and the goal of the project was to adhere to these best as possible (iOS Human Interface Guidelines, 2016). So as a fix, an alert asking for confirmation, that is triggered when player touches the quit button to leave the running game back to the menu screen, was added to the game.

When the application was deemed mature enough a more complete and planned user testing was performed. For this test, the supposed user group for the game was thought to be a young adult aged 18-35. This age group, according to a study, is the most active in playing mobile games, presenting 30% of mobile gamers. The same study indicates that the gender of the tester should not matter, as both men and womed seem to play almost as much. Because of the age group and that basic knowledge about smart phones is needed to finding and installing an application on phone, it was decided that the potential user has at least some experience with modern smart phones and mobile games. (Myth Busting: Mobile Gaming Demographics, 2015.)

The focus of this user testing was to find out if the game can be thought to be finished and how it could be further improved. Any possible problem in using the application also gives hints on how well iOS Human Interface Guidelines are fulfilled and thus, how App Store ready the project is: as was the main goal of the project. The user tests and its results are discussed further in chapter four.

2.3 Swift or Objective-C?

The first question when starting the project was which programming language to use. I had some experience with both Swift and Objective-C, which are used for writing iOS applications. Though the languages are used for same purposes, they differ a lot from each other.

Swift's syntax is generally simpler than what is found in Objective-C and this can make maintaining the code easier. A major feature in Swift is its optional system that guides programmer to do safer code. In both Objective-C and Swift values can be in nil state, meaning that value is empty. When something in code can be nil, Objective-C will not run it, nor will the compiler enforce the programmer to do anything about it. This behaviour can, however, lead to all kinds unpredictable errors and software bugs if one is not careful. In Swift its optional system forces unwrapping any possible nil values to make sure the value exists. It guides to write safer code where it is clear to see when a value can be nil and how the applications should react to it. This can also simultaneously be a drawback as Swift enforces a lot more rules for the developer than Objective-C.

```
import UIKit

class ViewController: UIViewController {

    var str: String?

    override func viewDidLoad() {
        super.viewDidLoad()

        guard let str = str else {
            return
        }

        print(str)
    }
}
```

Listing 1. Swift optional example

An example of how Swift's optionals can be used is shown in listing 1. In the example variable named str was declared optional with adding a question mark after its type, meaning that it will not need to be initialized with a string and that its value can be nil. When the ViewController loads, the variable is checked in guard statement, if it has a value it is copied to similarly named constant that overrides the name usage from then on and can be used safely. However, if the variable is nil, the guard-statement will return the function stopping the execution to avoid problems with nil values. It is also worth mentioning that Swift's print function can print optionals, meaning that the application would not actually crash without the guard statement. It is used purely as an example here.

Memory management has been completely automated with Automatic Reference Counting (also known as ARC) in Swift, unlike Objective-C where it has been only partially automated – mostly lacking in the low-level APIs. Swift has also been reported to be quite fast at running certain algorithms, even closing on C++ level of performance. (Solt Paul, 2015.)

Objective-C still has some merits such as it being a superset of C programming language makes interacting with C and C++ libraries much easier compared to Swift. Being an older programming language also means that there is much more discussion and info about it around. Also being older it is less subject to serious changes that would require huge amounts of refactoring the project's code.

Eventually, however, the more modern Swift was chosen for the project's programming language, mainly because of the overall cleaner code it can produce. A major issue caused by this decision was Swift's low maturity at the time. This led to large amounts of refactoring the code every time a new major version of Swift was released.

2.4 Game development frameworks for iOS then and now

Game development frameworks are split into two categories: there are low-level APIs that are meant to communicate with the hardware and then there are high-level frameworks or game engines that are built upon them. The line between what is a framework and a game engine is, however, blurry and mostly just down to semantics. (GameFromScratch.com 2015)

When iOS was opened for applications in 2008, the only way to create graphics and games was by using one of the supported OpenGL ES versions. However, OpenGL is a low-level API, meaning that creating anything is usually difficult and time consuming: even a simple tutorial to create a shaded box in OpenGL is pages long and involves a lot of mathematics to get it done (Wenderlich Ray, 2011). Because of that, several simpler to use frameworks have been built upon it. Even for iOS there are currently plenty of these high-level frameworks like Unity, Sparrow and Cocos2d-x. There can be a lot of differences between them, for example Unity and Cocos2d are cross platform but only Sparrow is programmed with Objective-C (Create games, connect with your audience, and achieve success, 2016; What is Cocos2d-x, 2016; Sparrow, 2016).

With iOS 7 Apple released its own high-level graphics frameworks: SpriteKit and SceneKit (Tabini Marco, 2013). SpriteKit is focused on 2D graphics while SceneKit focuses on 3D. These two can also be combined together: SpriteKit scenes can be used to create 2D overlays for SceneKit scenes and vice versa.

Apple has also released its own low-level API named Metal with iOS 8 that is meant to be more efficient than OpenGL ES. Apple is promising somewhere about 10 times the rendering speed with game engines built on Metal compared to OpenGL ES (Cohen Peter 2014). With iOS 9 both SpriteKit and SceneKit will use Metal for rendering as a default setting and some third party game engines such as Unity 5 also supports Metal (Apple, 2015; Zibreg Christian, 2015).

2.5 Best fitting framework for the project

Because of the simplistic game design that did not require any really special features, there really was not any reason to even start to consider creating an own game engine with a low-level API like Open GL ES or Metal. This would have also taken much more time and could have even been an overwhelming task for a one-person team to finish.

Discarding the low-level APIs left me with bunch of high-level framework choices and the choice between them was mostly just to personal preference. Game engines such as Unity would have made it easy to import the game to android and would have been more feature complete: as I was to find out when programming the game. Free version of Unity, however, comes with some handicaps and because the project had a zero

budget, this would have been the version I would have needed to use. (Create games, connect with your audience, and achieve success, 2016.)

Most of all I was also interested to use the new Swift language, which at the time was not in use in Unity. This excluded Unity from my consideration and left me with frameworks that can be used with Swift. From these I preferred Apple's own SceneKit as I saw a good opportunity to learn it and SpriteKit simultaneously, giving me a really good insight into Apple's own ecosystems.

2.6 Game project architecture

When coding the project I tried to minimize the amount of logic in view controllers to increase readability and possibility for re-use. The game has two view controllers: MenuViewController that manages the menu screen and GameViewController that is used for both of the games. The AppDelegate is used for persisting data that is shared between all the classes. This data includes the player's high scores for both of the games and sound settings. The data is persisted to <Application_Home>/Documents directory as is recommended by the iOS Data Storage Guidelines for things that can not be recreated by the application (iOS Data Storage Guidelines, 2016).

Because the project was started with swift 1.3 that did not yet have the capability to extend protocols with default behaviour, Swift's protocol system as is currently available is not utilised to its fullest. Instead traditional object oriented patterns were used.

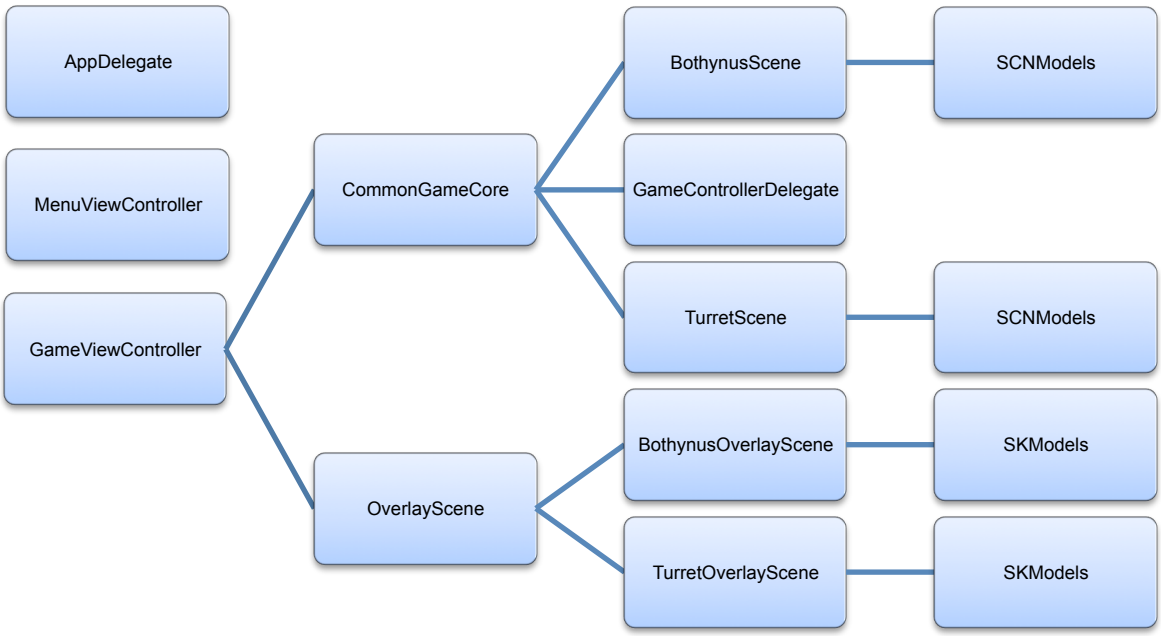

Image 2. Application architecture.

As illustrated in image 2, all the gaming related actions are run in GameViewController. To make it possible for it to communicate with two different games, there is a CommonGameCore named protocol that includes methods that both games share.

The GameViewController also holds a SCNView object that holds the SpriteKit overlay scenes that the game scenes use. The SceneKit and SpriteKit scenes then have their own related game models that they hold. The BothynusScene and TurretScene, shown in image 2, both subclass a SCNScene class, which will be talked more about later in this thesis.

```
internal protocol CommonGameCore: SCNSceneRendererDelegate,
SCNPhysicsContactDelegate {
    var gameStatus: Bool {get set}
    var gameControllerDelegate: GameControllerDelegate?
{get set}
    func handlePan(_ translation: CGPoint)
    func handleTap(_ location: CGPoint)
    func spawnMeteors()
    func updateRadar()
    func addParticleSystem(_ targetNodePosition: SCNVec-
tor3)
    func winRound()
    func gameOver()
    func removeNodes()
}

internal protocol GameControllerDelegate: class {
    func controllerWithCase(_ controllerCase: ControlCase)
}
```

```
internal enum ControlCase {
    case exitButtonPressed
    case audioControllerPressed
    case retryButtonPressed
}
```

Listing 2. CommonGameCore protocol, its methods and related delegate protocols and enumerations.

As shown in listing 2, CommonGameCore also implements two SceneKit related delegates. The SCNSceneRendererDelegate protocol gives access to SceneKits renderer methods that can be used for timed events, physics simulation and game logic, while SCNPhysicsContactDelegate protocol is for collision detection between the game models.

However, the biggest need for the CommonGameCore protocol is to make it possible to set the gameControllerDelegate property of a game scene when there are two different possible game classes, as was shown in image 2. The delegate also needs to be set to get information about any of the possible ControlCase cases show in listing 2. Without this the GameViewController would never know when to mute music or when to restart or quit the game.

One solution was to make both of the game scenes implement the CommonGameCore protocol, which has the gameControllerDelegate property, and cast to it in GameViewController. This way I could set the delegate property, making it possible for two slightly different games to share the same view controller and also to have the game logic out from the view controller in separate scene classes.

But why not just create a custom class that would subclass the SCNScene so no casting to protocol would be needed? Because of the game design, both of the games are a little different and thus, they differ also in their methods and what they do. To avoid unneeded clutter, this is an excellent case to use protocols. In GameViewController we only have to know that the needed variables and methods exist and not how they actually work. In this way it is possible to create methods that work differently but still have access to them.

Finally, the two different overlay scenes showed in image 2, both subclass a custom OverlayScene class, which in turn subclasses SKScene class, for all the shared meth-

ods. The overlay scenes then overrides them when needed. This too would have been cleaner to do purely with protocols that have their default behaviour extended, as overriding super classes methods can get a bit messy at times.

3 Developing with SceneKit

3.1 Basic knowledge about SceneKit

SceneKit works by having a hierarchy of SCNNode objects which can be made to display various graphics or animations, which can be either made with the many classes in SceneKit or they can be imported from external sources. (SceneKit, 2016.)

The SCNNode hierarchy works by adding these nodes as children to SCNScenes rootNode property that works as the base. These SCNNodes can themselves have any number of children and so on. Every SCNNode has its own local coordinate system meaning that moving a node that has child nodes will also make all the children move in relation to this movement. The same goes to adding and removing nodes: removing a node from scene will also remove all the children. (SCNNode, 2016)

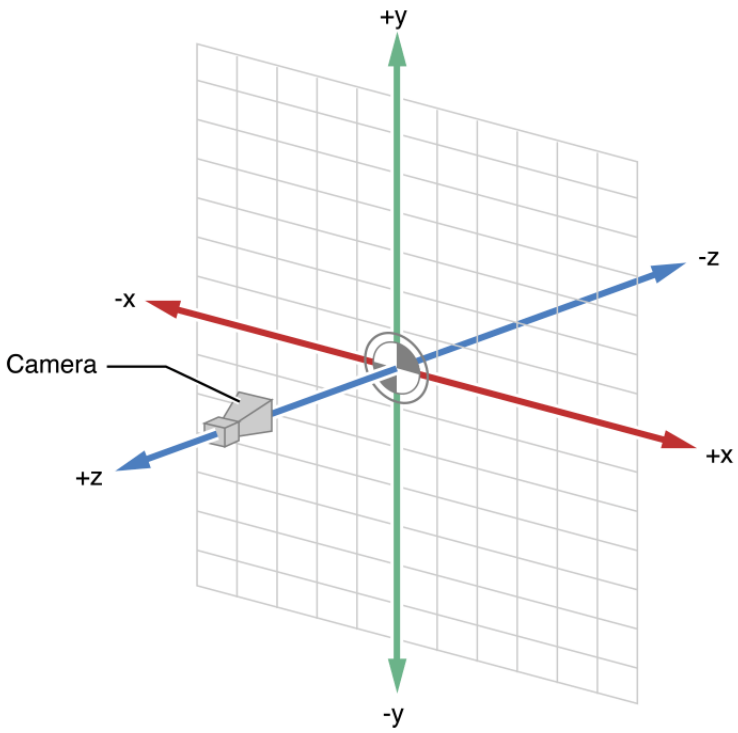

Image 3. SceneKits coordinate system. Copied from SceneKit (2016).

An important to know factor in SceneKit is the default direction that the camera will be looking. As is shown in image 3, the camera will face negative z-axis by default, x-axis will be to the sides as horizontal axis and y-axis will be the vertical axis.

SceneKit can also handle collision physics and detection, which were used extensively in the project. There are also multiple different objects for different kind of physics behaviours such as different kind of joints and vehicle behaviour. (SceneKit, 2016.)

3.2 Game view and game scenes

Creating the game with SceneKit started by adding a SCNView object to the GameViewControllers storyboard scene, this could also be done purely programmatically by adding it in the GameViewController by using its initWithFrame initializer: the frame in the initializer basically meaning the size of the wanted view object. (SCNView, 2016.)

SCNView subclasses NSView, meaning that it has the same properties and methods as NSView but with few SceneKit related additions. Most importantly it has a scene property that takes an SCNScene object. (SCNView, 2016.)

SCNScene objects are the base of a SceneKit games and animations. It has a rootNode property that has the node hierarchy included in the scene: nodes being all the objects and models in the scene (SCNScene, 2016). In the project there are two SCNScene objects, one for each game variant, they were created as separate classes that would then subclass the SCNScene. These are then added to the SCNView objects scene property in GameViewController according to which game was chosen by the player. There is, however, some dissonance between programmers on if the SCNScene should actually be subclassed or not: there are at least some reports of serialization not working correctly when subclassing it (Trouble subclassing SCNScene, 2015). However, as the project did not need serialization for the scenes, it was deemed worth having the game logic away from cluttering the GameViewController and to subclass SCNScene anyway. In the end this did not cause any issues in the game.

3.3 Game map and scenery

Because of the game design, creating the map was quite easy. I made a custom Map class that subclasses SCNNode, whose job is to create six Wall classes. These are then positioned and rotated to make a cube to restrict the playing area. The walls are

just simple invisible planes that are used to remove any SCNNodes colliding with them from the game.

```
import Foundation
import SceneKit

internal class Wall: SCNNode {

    internal init(position: SCNVector3, rotation: SCNVec-
tor4) {
        super.init()

        geometry = SCNPlane(width: 400, height: 400)
        physicsBody = SCNPhysicsBody.static()
        self.position = position
        self.rotation = rotation

        physicsBody?.categoryBitMask = CollisionCatego-
ry.Map
        physicsBody?.contactTestBitMask = CollisionCatego-
ry.All
        physicsBody?.collisionBitMask = CollisionCatego-
ry.All
    }

    internal required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
}
```

Listing 3. Wall class showing basic SCNode structure.

As shown in Image 3, the initializer for a Wall class creates a 400 by 400 plane that has physics body type of staticBody. Physics body needs to be set if collision detection with the SCNNode is desired. The staticBody type means that the wall will ignore all physics caused forces and can not be moved. The other possible physics body types are dynamicBody that is affected by all physics and kinematicBody that ignores any physics that would affect it but can cause collisions with others. (SCNNode, 2016.)

After setting the walls physicsBody property, it will be rotated and positioned as is decided by the Map class that is initializing the Wall class. Most importantly, no colours or textures are set to the wall, to make it invisible to the player.

The game design meant that it should look the player is in space, but the default black SceneKit background was deemed not good enough for this. Eventually a skybox with added stars was added to the game, by setting the SCNScene background content's property with an array of image paths. This background property will normally just take

one colour or image, which would then be shown always in the background but when the property is set with six images in an array, to make a cube, it will create skybox with them. (SCNScene, 2016.)

```
import Foundation
import SceneKit

internal class Earth: SCNNode {

    internal init(name: String) {
        super.init()
        self.name = name

        geometry = SCNSphere(radius: 10)
        physicsBody = SCNPhysicsBody.kinematic()

        geometry?.firstMaterial?.diffuse.contents =
UIImage(named: "art.scnassets/earthmap1k.jpg")
        geometry?.firstMaterial?.emission.contents =
UIImage(named: "art.scnassets/earthlights1k.jpg")
        geometry?.firstMaterial?.specular.contents =
UIImage(named: "art.scnassets/earthspec1k.jpg")
        geometry?.firstMaterial?.diffuse.mipFilter = SCN-
FilterMode.linear
        geometry?.firstMaterial?.emission.mipFilter = SCN-
FilterMode.linear

        physicsBody?.categoryBitMask = CollisionCatego-
ry.Earth
        physicsBody?.contactTestBitMask = CollisionCatego-
ry.All
        physicsBody?.collisionBitMask = CollisionCatego-
ry.All


runAction(SCNAction.repeatForever(SCNAction.rotate(by:
CGFloat(M_PI_4), around: SCNVector3(x: 0, y: 1, z: 0), du-
ration: TimeInterval(10))))
    }

    internal required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
}
```

Listing 4. Earth class showcasing SceneKit material property usage.

The main game also needed an earth-like planet that would then be bombarded by meteors. The planet was created with simple sphere geometry, but as shown in Image 4, with added SCNMaterial properties. SCNMaterial is what gives SCNGeometry its actual rendered appearance while the SCNGeometry only gives the shape (SCNMaterial, 2016).

For the Earth, several of these material attributes were used. Normal Earth texture was used for the diffuse that is shown on the parts that get light and emission texture of dark Earth for the parts that do not. The emission material will glow in the dark and will not need light source to make the texture visible. This creates more realistic Earth that can also be distinguished from the background when the player is looking it from the dark non-lighted side. Specular map was then added, making only the seas reflect light while ground and mountains would absorb it. Specular map is basically just black and white picture where black parts will not reflect light while white parts will.

Both the emission and diffuse materials also had their mipFilter property set to linear. This was made for added rendering performance and reduction in rendering artefacts that could appear when moving closer to the model (SCNMaterialProperty, 2016).

A Cloud class that is a nothing but a simple white sphere was then added on top of the Earths model. The clouds use SCNMaterials transparent property with a transparency map to make parts of it invisible, now only showing parts of the white sphere as clouds and the Earth layered below between them.

Finally, both the Earth and the clouds are made to slowly rotate by their y-axis, with a repeated SCNAction, which will be discussed about later in this thesis. This is to make the map seem more living and as a work a round for a problem involving the shopping screen, that is shown with overlay scene, stopping rendering its animations when there was nothing moving in the actual game scene

SCNMaterial has also many other properties such as a possibility to add a normal or bump map that would make Earths Mountains actually stand up from the sphere. Or make parts of the seas always reflect more light, no matter the lights direction with ambient property. These were however not added to the game, because of the lack of proper maps to add them. (SCNMaterial, 2016.)

The game scenes are lighted by a SCNLight that has four different possible types. There is ambient type that lights everything ignoring its position and direction, an omni that will shed light to all directions from it’s position, direction type that will light everything from certain direction ignoring the lights position and finally there is a spot lighting type that will only light objects at its position and direction. For the game design the omni light type was deemed best as it simulates sun the best. (SCNLight, 2016)

Basic SCNLight setup works by setting SCNNodes light property with a SCNLight and then setting the SCNLights type-property with the desired type. This was also done in the project. The SCNNode with the light was then positioned behind the player, so Earth and most of the meteorites would be visible to the player in the start to make the start more accessible. (SCNLight, 2016)

3.4 Building a flying first person shooter controller

The game design stated that the game needed a first-person shooter controller and that the controller should rotate by swiping and shoots by tapping. Rotation was meant to be handled in inverted fashion, where the player's direction will change to the opposite of the swiped direction. In the main game player will move static amount every render loop and the speed is decided by the amount of speed-boosts purchased in the game. This design solves the problem of not having more natural ways of distinguishing different kind of user inputs, unlike in computer games where keyboard can be used for movement and mouse for rotation.

```
let panGesture = UIPanGestureRecognizer(target: self, ac-
tion: #selector(GameViewController.panGesture(_:)))
view.addGestureRecognizer(panGesture)

let tapGesture = UITapGestureRecognizer(target: self, ac-
tion: #selector(GameViewController.handleTap(_:)))
view.addGestureRecognizer(tapGesture)
```

Listing 5. Example of how to add gesture recognizers to view controller.

To get the swipe and tap input information gesture recognizers were added to GameViewController as shown in Listing 5. The action fields specify what methods should be called when the specific gesture is recognized. Both of the recognizers have their own methods specified in their appropriate classes.

```
func panGesture(_ gesture: UIPanGestureRecognizer) {
        guard let gameScene = gameScene as? CommonGameCore
, gameScene.gameStatus else {
            return
        }
        let translation = gesture.translation(in:
self.view)
        gameScene.handlePan(translation)

}
```

```
func handleTap(_ sender: UITapGestureRecognizer) {
    if sender.state == .ended {
        guard let gameScene = gameScene as? CommonGameCore
else {
            return
        }

        let location = sender.location(in: self.view)
        gameScene.handleTap(location)
    }
}
```

Listing 6. Example of handling recognized pan and tap gestures in a view controller.

Listing 6 displays the UIPanGestureRecognizers panGesture and UITapGestureRecognizer handleTap methods specified in listing 5. The listing also displays the reason of needing a common protocol that is shared by both of the different game scenes. It made it possible to cast the currently used game scene to the protocol and gain access to a method that is now known to exist. This can then be used to share translation information of the swipe to the scene, which will use it to rotate the player accordingly.

3.4.1 Rotation and Euler angles in SceneKit

SCNodes can be rotated by several different ways. SCNodes have rotation, eulerAngles, orientation and transformation properties that can all be used to rotate the node. They are all connected in a way where if one is set, all the other properties are also changed to reflect the new rotation.

The simplest way to rotate a SCNNode is by setting the rotation property with a four-component vector. The three first values of this vector tell the rotation direction and the fourth value gives the amount of rotation. This would have been enough for the turret game, as setting the rotation property worked well enough with just one rotation axis. However, this did not work well for the main game where the player needs to be able to rotate in multiple different dimensions this.

Step up from this was to use the Euler angles. Euler angles have three components: pitch, yaw and roll where pitch is the rotation on x-axis, yaw on y-axis and roll on z-axis. Together they can be used to represent rotation in multiple dimensions, which the controller for the main game needed.

Using Euler angles worked until it collided with a problem known as gimbal lock. This happens when one of the three components is rotated by pi / 2 radians or 90 degrees so that it will line up with one of the other components, thus loosing one degree of freedom. When this happens the SCNNode would lose its sense of rotation in the three dimensions, which leads to weird behaviour such as the camera quickly turning to opposite direction and back as it won't anymore know which direction it should be facing. This meant that either Euler angles weren't the best choice for the requirements.

3.4.2 Quaternions and how to use them

A solution for handling rotation in three dimensions without any gimbal lock problems was to use SCNNodes orientation property that uses quaternions. Quaternions however are quite a bit more complex to use, as SCNQuaternion class did not have any required quaternion math methods. To solve this deficient I either needed to create the methods by myself or use GLKits GLKQuaternion class. GLKQuaternion has all the needed math functions and the calculated results components can be used to create the SCNQuaternion that can set the SCNNodes orientation. For this project creating the functions by myself was deemed better for learning purposes and a way to avoid unnecessary GLKQuaternion to SCNQuaternion conversions.

```
func handlePan(_ translation: CGPoint) {
    let orientation = playerNode.presentation.orientation
    let position = playerNode.presentation.position

    if fabs(translation.x) > fabs(translation.y) {
        let rotation = Float(M_PI / 180) *
Float(translation.x / 50)

        let convertedAxis = multiply(multiply(orientation,
q2: SCNQuaternion(x: 0, y: 1, z: 0, w: 0)), q2: conju-
gate(orientation))
         playerNode.rotatePlayer(convertedAxis.x, y: con-
vertedAxis.y, z: convertedAxis.z, angle: rotation)

         let newOrientation = play-
erNode.presentation.orientation
         let direction = multiply(multiply(newOrientation,
q2: SCNQuaternion(x: 0, y: 0, z: -1, w: 0)), q2: conju-
gate(newOrientation))

         var angle = atan2(-1, 0) - atan2(direction.z, di-
rection.x)

         if angle < 0 {
             angle += Float(2 * M_PI)
         }
```

```
            overlayScene.radarPlayerRotation = CGFloat(angle)

        } else {
            let rotation = Float(M_PI / 180) *
Float(translation.y / 50)
            let convertedAxis = multiply(multiply(orientation,
q2: SCNQuaternion(x: 1, y: 0, z: 0, w: 0)), q2: conju-
gate(orientation))
            playerNode.rotatePlayer(convertedAxis.x, y: con-
vertedAxis.y, z: convertedAxis.z, angle: rotation)
        }

        playerNode.position = position
    }

    internal extension Player {
        func rotatePlayer(_ x: Float, y: Float, z: Float, an-
gle: Float) {
            let tempQ = SCNQuaternion(x: x * sinf(angle / 2),
y: y * sinf(angle / 2), z: z * sinf(angle / 2), w:
cos(angle / 2))
            orientation = multiply(tempQ, q2: presenta-
tion.orientation)
        }
    }


    internal func multiply(_ q1: SCNQuaternion, q2: SCNQuater-
nion) -> SCNQuaternion {

        let w = q1.w * q2.w - q1.x * q2.x - q1.y * q2.y - q1.z
* q2.z
        let x = q1.y * q2.z - q1.z * q2.y + q1.w * q2.x + q2.w
* q1.x
        let y = q1.z * q2.x - q1.x * q2.z + q1.w * q2.y + q2.w
* q1.y
        let z = q1.x * q2.y - q1.y * q2.x + q1.w * q2.z + q2.w
* q1.z

        return SCNQuaternion(x: x, y: y, z: z, w: w)
    }

    internal func conjugate(_ q: SCNQuaternion) -> SCNQuaterni-
on {
        return SCNQuaternion(x: -q.x, y: -q.y, z: -q.z, w: q.w)
    }
```

Listing 7. Rotation handling and quaternion helper methods.

Listing 7 displays all the functions used for rotating the player. Firstly the GameViewController calls the handlePan method and then the translation is checked if the player wants to rotate in vertical or horizontal direction. The translation amount in this check needs to be an absolute value, because depending on the direction the value might be negative. Then the rotation value is adjusted by an arbitrary amount, to make the player rotate in a desired speed.

Depending on the rotation direction, either a normal vector representing the z-axis or y-axis is then converted from global axis to local by multiplying it with the player's current orientation. To get the SCNNodes current orientation it's needed to use its presentation property's values. The presentation always displays the current properties while calling the SCNNodes properties directly always give the ones it was set with. The presentation is also a read only property, so these can not be used to set the rotation. The calculation operation for this is qpq^-1, where q is the quaternion that rotates the quaternion p and q^-1 is conjugate of quaternion q. The order in this operation matters and multiplying qp gives different result than pq. The way these two operations work can be observed from listing 7 from their similarly named methods. In this case this is used in a way where first the player's current orientation is multiplied with normal vector representing a wanted axis of rotation using the self-made multiply operation. Then this is multiplied with the conjugate of player current orientation, as is shown in listing 7. (Jeremiah van Oosten, 2012.)

The x, y and z components of this quaternion are then passed to the playerNode with the amount to rotate. The rotatePlayer method then calculates a new quaternion that can be used to rotate the player by multiplying these two together. These calculations required a creation of a temporary quaternion named tempQ that converts from axis-angle to quaternion. This converted value can then be used to rotate the player correctly.

Finally as displayed in listing 7 the player's position is reset with position it had before changing its orientation, because the players transformations position is reset to its starting position when orientation is set.

3.4.3 Adding the shooter element to the controller

Game design required making it possible for the player to shoot by tapping the screen. This was done by using the information given by handleTap-method that was shown earlier in listing 6. The information on where the player tapped is not important in this part, because the ammunition should just shoot directly forwards from the player.

```
if hp > 0 && gameStarted && ammo > 0 {
    let orientation = playerNode.presentation.orientation
    let direction = multiply(multiply(orientation, q2: SCN-
Quaternion(x: 0, y: 0, z: -1, w: 0)), q2: conju-
gate(orientation))
```

```
    let bullet = Bullet(position: SCNVector3(x: play-
erNode.presentation.position.x + direction.x, y: play-
erNode.presentation.position.y + direction.y, z: play-
erNode.presentation.position.z + direction.z))
    bullet.physicsBody?.applyForce(SCNVector3(x: direc-
tion.x * 20, y: direction.y * 20, z: direction.z * 20),
asImpulse: true)
    rootNode.addChildNode(bullet)
    ammo -= 1
}
```

Listing 8. Example of controllers shooting method.

First an if-clause was added to limit shooting to only when player is still alive, has ammunition and the game is running. To get the shooting direction, quaternions were used to rotate the player starting direction vector that is facing -1z by the players presentations orientation, as shown in listing 8. Then a Bullet class, that has dynamic physicsBody type to make them influenced by all the physics, is created to the player's presentations current position.

The bullet can be pushed forward by applying force to its physicsBody-property with a vector created from the direction like is done in listing 8. To make the bullets fly faster, the values of this vector are multiplied by arbitrary amount.

3.5 Game models

The simplistic game design required only a few visible models that needed to be displayed. The Earth and bullets were fully created with the tools provided by Xcode, I only needed the textures and maps for the Earth and its clouds.

```
extension Bullet {
    func particleSystem() {
        guard let particleSystem = SCNParticleSystem(named:
"BulletParticle", inDirectory: nil) else {
            return
        }
        particleSystem.emitterShape = geometry
        addParticleSystem(particleSystem)
    }
}
```

Listing 9. Adding a particle system to bullet.

The bullets were created as just a small white spheres that have custom made particle effect system attached to them. The used effect was designed with Xcodes particle system file generator that can be used to create 3D particle systems for SceneKit easily. Xcode also has its own generator for SpriteKit that does 2D particle systems. This generated file is then imported to the game by initializing a SCNParticleSystem with it, like was done in listing 9. This SCNParticleSystem is then given the shape of the bullet sphere so the particle effect will surround the bullet nicely and finally this system is added to the bullet as a child node. This makes the system follow the bullets position and to get removed from the scene and memory when the bullets are.

The meteors and comets, however, required custom made 3D models that were created with Blender. To add these to the project the models were first exported from blender as COLLADA file, which is a XML schema holding all the information about the model. This file, with .dae extension, can then be imported with any needed textures to Xcode.

```
import Foundation
import SceneKit

internal class Meteor: SCNNode {

    internal init?(position: SCNVector3, randRot: Int) {
        super.init()

        name = "meteor"
        guard let url = Bundle.main.url(forResource:
"art.scnassets/meteor", withExtension: "dae"), let source =
SCNSceneSource(url: url, options: nil), let geometry =
source.entryWithIdentifier("Cube-mesh", withClass: SCNGeom-
etry.self) as SCNGeometry? else {
            return nil
        }

        self.geometry = geometry
        physicsBody = SCNPhysicsBody.kinematic()

        let material = SCNMaterial()
        material.lightingModel = .lambert
        material.diffuse.contents = UIImage(named: "RockS-
mooth.jpg")
        material.diffuse.wrapS = .repeat
        material.diffuse.wrapT = .repeat
        material.diffuse.intensity = 0.5
        material.emission.contents = UIImage(named: "RockS-
mooth_dark.jpg")
        material.emission.wrapS = .repeat
        material.emission.wrapT = .repeat
        material.emission.intensity = 0.5

        self.geometry?.replaceMaterial(at: 0, with: materi-
al)
```

```
        scale = SCNVector3(x: 3, y: 3, z: 3)

        physicsBody = SCNPhysicsBody.kinematic()
        physicsBody?.damping = CGFloat(0)
        physicsBody?.categoryBitMask = CollisionCatego-
ry.Meteor
        physicsBody?.contactTestBitMask = CollisionCatego-
ry.Bullet | CollisionCategory.Player | CollisionCatego-
ry.Earth | CollisionCategory.Comet
        physicsBody?.collisionBitMask = CollisionCatego-
ry.Bullet | CollisionCategory.Player | CollisionCatego-
ry.Earth | CollisionCategory.Comet

        self.position = position
        rotation = SCNVector4(x: 0, y: 1, z: 0, w:
Float(M_PI) / Float(8) * Float(randRot))
    }

    internal required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
}
```

Listing 10. Example of a Meteor class initialisation.

To add this imported COLLADA file to a SCNNode, it is needed to create a SCNSceneSource with the files URL that can be acquired from the projects main bundle. This source can then be used to initialize a SCNGeometry, by using the SCNSceneSources entryWithIdentifier method, the identifier being the name of the models mesh. The result then needs to be casted to SCNGeometry, so it can be used to set the Meteors geometry property like was done in listing 10.

All these return optional values, so they were chained in guard statement to make sure the game will not crash if something goes wrong. This guard will make the failable initializer return nil in case of error, which will then be discarded in the game when meteors are spawned.

Listing 10 also show a workaround for problem that arose when the code was refactored from Swift 2.2 to Swfit 3. The texture setting that were included in the COLLADA file wouldn't no longer successfully be applied automatically when the model was imported. As a quick solution, a SCNMaterial class with the same attributes that the COLLADA would've had is created and added to the SCNNode. This included setting the materials lightningModel property to Lambert that decides the lightning formula used for rendering, setting the wrap properties to repeat the texture and finally setting the intensity property to 0.5 so to not make texture too bright.

A problem related to these models was found in testing. Because of the game design that wanted to simulate the light sun gives, with the omni-type SCNLight, the meteors were hard to see from the dark side. As a solution, a darker version of the used texture is also set to the meteors materials emission, as is shown in listing 10. This makes the meteors more visible, no matter the direction the player is looking at the models.

To make the imported models larger the SCNNodes scale property is set to make them three times larger in all dimensions. Finally the meteors are rotated by random amount, to make the spawned meteors all face different direction.

The comet models use the same COLLADA models as meteors but are scaled to even larger size. They also have an added particle system that is added by a method described earlier in listing 9. This not only makes them look more menacing but also easy to distinguish from others.

3.6 Animating in SceneKit

For doing animations, in SceneKit there are couple of choices. Externally created animations can be imported as COLLADA files and these can be used to initialize an SCNScene object that can then be used as desired (SCNScene, 2016). Purely code-based animations can be handled per frame basis in rendering loop or by creating SceneKits own SCNActions or SCNTransaction. Animations created in Core Animation frameworks can also be added to SceneKit by using SCNActionable interface.

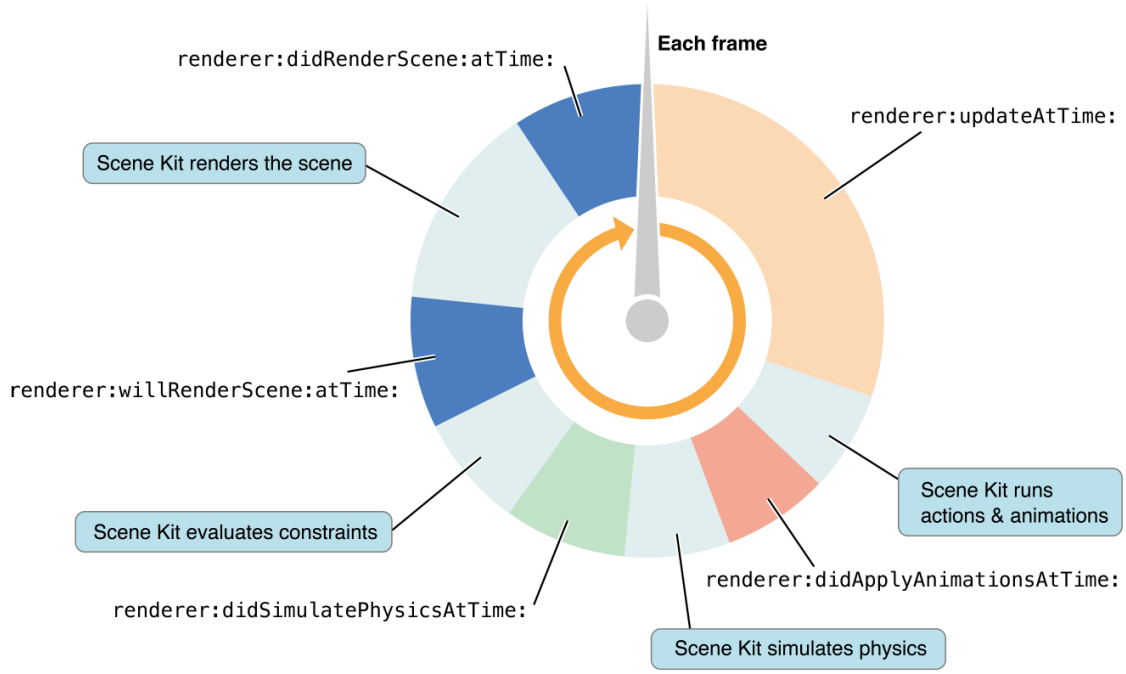

Image 4. SceneKit rendering loop and its delegate. Copied from SCNSceneRendererDelegate (2016).

To do per frame animations in SceneKit a class has to implement SCNSceneRendererDelegate protocol and be set as delegate for the SCNView that is currently used for the rendering. The delegate can implement any of the methods listed in image 4 for different phases of the render loop that are ranging from the start of the update loop to the moment when the scene is fully rendered. Importantly these methods should not contain overly complex and time-consuming logics as it can slow the rendering time. (SCNSceneRendererDelegate, 2016.)

```
func renderer(_ renderer: SCNSceneRenderer, updateAtTime
time: TimeInterval) {
    if gameStarted {
        let ppo = playerNode.presentation.orientation
        let direction = multiply(multiply(ppo, q2: SCNQua-
ternion(x: 0, y: 0, z: -1, w: 0)), q2: conjugate(ppo))
        playerNode.physicsBody?.velocity = SCNVector3(x:
direction.x * Float(8 + shipSpeed), y: direction.y *
Float(8 + shipSpeed), z: direction.z * Float(8 + ship-
Speed))

        if cometResponseTime >= 300 {
            cometResponseTime = 0
            for comet in cometNodes.childNodes {
                let vx = comet.presentation.position.x -
playerNode.presentation.position.x
                let vy = comet.presentation.position.y -
playerNode.presentation.position.y
```

```
                let vz = comet.presentation.position.z -
playerNode.presentation.position.z

                let mVector = sqrt(powf(vx, 2) + powf(vy,
2) + powf(vz, 2))

                comet.physicsBody?.velocity = SCNVector3(x:
-vx / mVector * Float(10 + difficulty), y: -vy / mVector *
Float(10 + difficulty), z: -vz / mVector * Float(10 + dif-
ficulty))

            }
        } else {
            cometResponseTime += 1
        }
        updateRadar()
    }
}
```

Listing 11. SCNSceneRendererDelegate example.

As demonstrated in listing 11 SCNSceneRendererDelegates upDateAtTime method was chosen for the project. This method is called in the beginning of a render loop, before anything has been drawn (SCNSceneRendererDelegate, 2016). Listing 11 also shows some of these animations that are done in the loop. The player is always moved forwards in its current direction by setting the velocity of playerNodes physicsBody when the game is running. Every frame, the player's current orientation is calculated by using quaternions in the same way as was done with the shooting. Because handling the player's rotation is separate method from this, checking the players orientation in every render loop is easiest and safest way to make sure the movement direction is always correct before any actual movement is rendered.

The comets, that give chase to the player, are also animated in the render loop. Their homing capability is however handicapped in a way that they will calculate their direction to the player only once per 300 renderer loops. This means a response time of 5 seconds when the game is running in its full 60 frames per second rendering rate. The handicap makes it possible for the player to do evasion movements to avoid collision with the comets, who by default will be moving at higher velocity than the player.

The movement direction for comets is determined by first calculating the needed x, y and z vector components, by reducing the point of playerNodes presentations position from the point of comets presentations position, as was done in listing 11. These components are then combined and normalized to create a normal vector pointing from the comet to player.

Because of the game design, the movement speed of the comets is increased every successive game round and players movement speed can be increased by purchasing more ship speed. This makes for an ever-increasing cat and mouse game that was intended. Though it's also marginally possible for the player to just keep trying to evade the comets if they get faster than the player. Constant evasion movements will, however, make it hard to aim and shoot all the meteors in time, meaning that the player can't ignore purchasing more speed boosts forever.

SCNSceneRendererDelegate is also used to refresh the radar that is displayed to the player, to keep its information correct. Not shown in listing 11 are also several warning that can be displayed to the player such as if the player is getting too close to the invisible map walls or if comets are getting dangerously close. These warnings have simple Boolean locks that make sure they are only launched once in an occasion or otherwise they would be displayed to the player multiple times in a second.

3.7 Using SCNActions

Rest of the animations in the projects were simple enough that SCNActions could be used create them. SCNActions have limited amount of capabilities mostly ranging from moving, rotating, scaling and chancing the alpha of objects. These can, however, be combined together by grouping and sequencing them, to create surprisingly complex animations. The animations can also be repeated from finite to infinite times as needed. Most importantly SCNActions take in blocks that can execute whatever is needed, making it possible to create custom actions. Even if SCNActions are categorized as a simple way to animate objects in can be used to make even complicated animations. (SCNAction, 2016.)

```
func shieldMeteorMovement(_ sp: SCNVector3, ep: SCNVector3)
{
    runAction(getMeteorAction(position, ep: SCNVector3(x:
0, y: 0, z: 0), TurretSpeedUp: TurretSpeedUp))

    let shieldAction = SCNAction.sequence([
        SCNAction.wait(duration: 1),
        SCNAction.run({(node) -> Void in
           node.physicsBody?.categoryBitMask = Collision-
Category.ShieldMeteor

           node.geometry?.firstMaterial?.reflective.contents
= UIColor.red
```

```
            }),
        SCNAction.wait(duration: 0.5),
        SCNAction.run({(node) -> Void in
                node.physicsBody?.categoryBitMask = Colli-
sionCategory.Meteor

           node.geometry?.firstMaterial?.reflective.contents
           = UIColor.black
            })
            ])

    runAction(SCNAction.repeatForever(shieldAction))
}

func getMeteorAction(_ sp: SCNVector3, ep: SCNVector3) ->
SCNAction {
    let vectorX = ep.x - sp.x
    let vectorZ = ep.z - sp.z
    let magnitudeVector = sqrtf(pow(vectorX, 2) +
pow(vectorZ, 2))

    return SCNAction.move(to: ep, duration: TimeInter-
val(Float(0.25) * magnitudeVector))
}
```

Listing 12. Example case of SCNAction usage.

Most of the SCNActions in the game are related to moving the meteors like in listing 12. The shieldMeteorMovement method will first call a helper getMeterAction method to create action that will move the meteor from its current position towards the Earth that is positioned at the origin of the scene. The SCNActions move-animation requires information on how long the movement should take, so to get all the meteors to move at same speed, a magnitude vector from the movements' starting and ending position is calculated. This magnitude vector is then used make the movements duration be in relation to the distance the meteor is from Earth, making them all move at same speed.

Afterwards in listing 12, a SCNAction sequence is made that is then set to be repeated forever until the node is removed. The sequence will first wait for one second to run a custom block that will change the meteor reflection colour to red and change its bit mask property. Then the sequence waits for half a second to change the reflection colour and bit mask back to what they were in the start. The intention of this is to make the meteors invulnerable for half a second as was mentioned in the game design. The other meteor types have similar kind of SCNAction usage: the normal meteors only need to be moved to origin in straight line, while the meteors that will move in more unpredictable path have a sequence of different points to move to.

3.8 Collision detection

To successfully implement the game design, some kind of information about when bullets hit meteors, meteors hit Earth or comet hit player was needed. Meaning that collision detection had to be added to the game. This can be done in SceneKit by setting a SCNScene class as a SCNPhysicsContactDelegate. Implementing this protocol gives access to three different optional methods that give info about when contact begun, was updated or ended. These methods will return the contact that happened, which in turn can be asked about the position of contact and what were the two nodes involved. (SCNPhysicsContactDelegate, 2016.)

Because in the game there can be multiple kinds of contacts and they all should be handled differently, so a way to distinguish what kind of nodes made the contact was needed. SCNNodes have a categoryBitMask property that needs to be set with a integer unique to the type (SCNNode, 2016). The nodes physcisBody property has contactTestBitmask and collisionBitMask properties that should be set with the types that can cause collision with the said SCNNode. This is done to limit the amount of collisions that will be detected. Listing 10 earlier shows how this was done in the project: firstly all the bit masks were made static constants, so they could be accessed globally, then these were applied to the listed properties accordingly.

To actually find out what kind of nodes caused the contact, the categotyBitMask property can be asked from the contact causing nodes in the delegate methods. These can then easily be used to compare what kind of SCNNodes they were and then act as required. In this project this involved a long switch that compares the nodes.

3.9 Adding SpriteKit overlay

The game needed some smart way to give information about the score, hit points, ammunition count, radar and to show messages to the player. Also, a way to display the shopping screen after player finishes a round was required. One easy way to add these in SceneKit is by adding a SpriteKit overlay to the SCNView. This can be done by setting the overlaySKScene property of SCNView with a SKScene object (SCNView, 2016).

The game projects architecture proved slightly cumbersome when this was done, as both the SCNView and SCNScene objects needed a reference to the overlay scene while the SCNView would also hold reference to the SCNScene. This could potentially cause a memory leak in the game, when two separate entities share strong reference to same object. Removing the overlay scene reference from SCNView before SCNScene is removed solved this potential problem. Overall it might be worth improving the architecture to avoid this kind of bad and risky patterns.

The overlay classed listed earlier in image 2 subclasses an SKScene class that is the SpriteKit equivalent of SCNScene class. So the radar, shopping screen and all the information labels were created as a SpriteKit objects that were then added to the SKScene classes as child nodes. These are then either hidden or shown depending on the games state, so for example, as was shown earlier in listing 11 the radar is only updated when the game is running and not when the shopping screen is shown. The shopping screen uses the same handeTap method that the controllers shooting action is using, to distinguish when and what shopping screens buttons were tapped. So this time also the information of what point in the screen the player tapped was needed. This information could easily acquired by first converting the point from ViewnControllers coordinate system to SpriteKits, as this two have different origin points, by using SKScenes converPoint method. The SKNodes in the overlay scene could then be asked if they contained this point, by using their contains-method that returns an appropriate Boolean value.

More extensive inner workings of these overlay scenes are out of the scope of this thesis that is focused on SceneKit and not in SpriteKit. However, overall the SpriteKit framework and its classes worked very similarly to SceneKit and its equivalent classes: just in 2D instead of 3D. This meant that using SpriteKit for the overlay scenes did not require much extra learning and by knowing how SceneKit worked, I was pretty much good to go.

4 Testing

4.1 Testing procedure

As was discussed in chapter 2.2.2, the most likely user group for the game was decided to be between the ages of 18 and 35, as this was the largest mobile gamer group. This meant that the testers would preferably be of this age group. It was also decided that the testers gender would not matter. However, the tests that were carried out had clearly more men than women.

The testing was meant to be light and manageable but it was also meant to give clear signs of how the finished game feels for a user. Further feedback on how the game could be improved was also needed.

In the tests, a tester was given an iPhone 5s that was preloaded with the game as a test device and he or she was asked to play the game couple of times. The device was chosen because of its smaller screen size compared to the newer iPhone models, so it could be known that the graphics can be seen clearly even on a small screen. The device also has worse processing power than the newer ones so if the game runs well on iPhone 5s, it should also run well on newer models.

After the testers felt confident enough to answer related questions about the game they were asked three questions and the main points of their answers were written down. The questions in English were “What did you like in the game?”, “What didn’t you like in the game?” and “How would you improve the game?”.

4.2 Testing results and analysis

A total of six people were interviewed for the user testing ranging from the age of 23 to 30. Each tester had some experience with mobile games, so any improvement ideas were thought to be highly valuable. The gender distribution of these six testers was five males and one female, though as stated, this was not deemed important. Everything was done anonymously, only the testers age was asked to make sure that they fit the target user group.

Testers age in years	What did you like in the game?	What you didn't like in the game?	How would you improve the game?
23	Liked graphics.	Controls were confusing.	Would add instructions on how controls work
27	Liked graphics.	Controls were confusing.	Would add instructions on how controls work and more manuals overall
30	Liked graphics and the games idea.	Controls were confusing.	Would add instructions on how controls work. Would also reduce explosion particles when close to one.
27	Liked graphics.	Controls were confusing.	Would add instructions on how controls work
27	Liked graphics.	Controls and the games objective were confusing.	Would add instructions on how controls work and make the controls slower. Would also add more settings to the controls and change some of the games terminology such as "HP" to "Life"
23	Liked the games idea.	Controls and the games objective were confusing.	Would add more manuals.

Table 1. User testing results.

Because of the clear results listed in table 1, the user testing sample size was thought to be enough as there was already an obvious pattern that could be analysed. This also meant that the test was successful in its objective of giving information of what still needs to be enhanced in the project.

The user test results shown in table 1 are very clear on what has been done right and what still has to be improved. All the testers answered basically the same things. The graphics are fine and the game looks good but the controls still need clear instructions how they work. The game could also have a clear manual what the player should be doing in it, instead or in addition to small text paragraph that is currently shown before the game starts. There are also some more optional tweaks such as chancing the terminology used in the game and making the explosion particle effects block less when they are triggered right next to the player. These too are, however, worth looking into, as they are trivially easy and fast to fix.

Though it could be argued, if these issues might break the human interface guidelines as they are more of a problem with the game design. However, they need to be corrected, before the project can be deemed completely finished and ready to be published in App Store.

5 Project Summary

5.1 Summary of what was done

Overall the project was mostly successful: all the game design specifications were met, but reaching the main goal still requires the addition of more instructions. The game has two different game modes: one where the player can move and one where the player is stationary. Though the games only had small differences, the differences were enough to try different kind of patterns, and try to limit unnecessary code repetition.

Using swift as a programming language made the code much more readable and maintainable compared to Objective-C. Swift's much safer type system makes sure that the program will not crash because of nil pointer errors. Because it is easy to know when some return value can be nil, it was easy to program the application so that it can resolve these issues without the whole application crashing.

The planned architecture for the game did not fully prove itself, as there still are some slight code repetition and possibility for memory leaks if not careful. Still overall, using protocols for common methods made it possible for handling the two different games in one ViewController class.

The game includes a custom-made first person shooter controller for a device with touch screen. The controller uses quaternions for calculating the player's orientation correctly as other methods of doing this could not handle rotation in multiple dimensions well enough. SceneKits built-in methods for using quaternions are not as easy to use as it might be in frameworks such as Unity, but these shortcomings can be alleviated either by using GLKits quaternion methods or with some math knowledge these can be implemented by self. The user testing however revealed that without instructions these controls can be confusing at first for the potential user.

The design required several different models, which are either done with SceneKits built in features, imported from COLLADA files that were made with Blender or the combination of these. SCNNodes have several properties that can be changed to make their materials colour and texture look as wanted and these were utilised in the project almost fully. Only the lack of good UV maps meant that these were not used for the Earths model.

SceneKit had multiple ways of animating these models, however, most of the possible methods were not needed in the project. Even the most simple of these, SCNActions, proved more than capable of handling even the more complex animations with ease. The SCNRenderDelegate gave an easy way to attach animations to the frame-rendering loop. This was also used to use simple timed actions that should only be triggered when certain conditions were matched. SCNRenderDelegate also had multiple different possible methods for different points of the renderer loop, which the first one that is triggered right at the start was found most usable for the project.

All the needed physics could be done with SceneKit and it was easy to configure which objects can bounce and which can not when colliding with other objects. The collision detection was also fairly easy to setup, though the current implementation required quite long and precise switch statement that handles all the possible cases correctly.

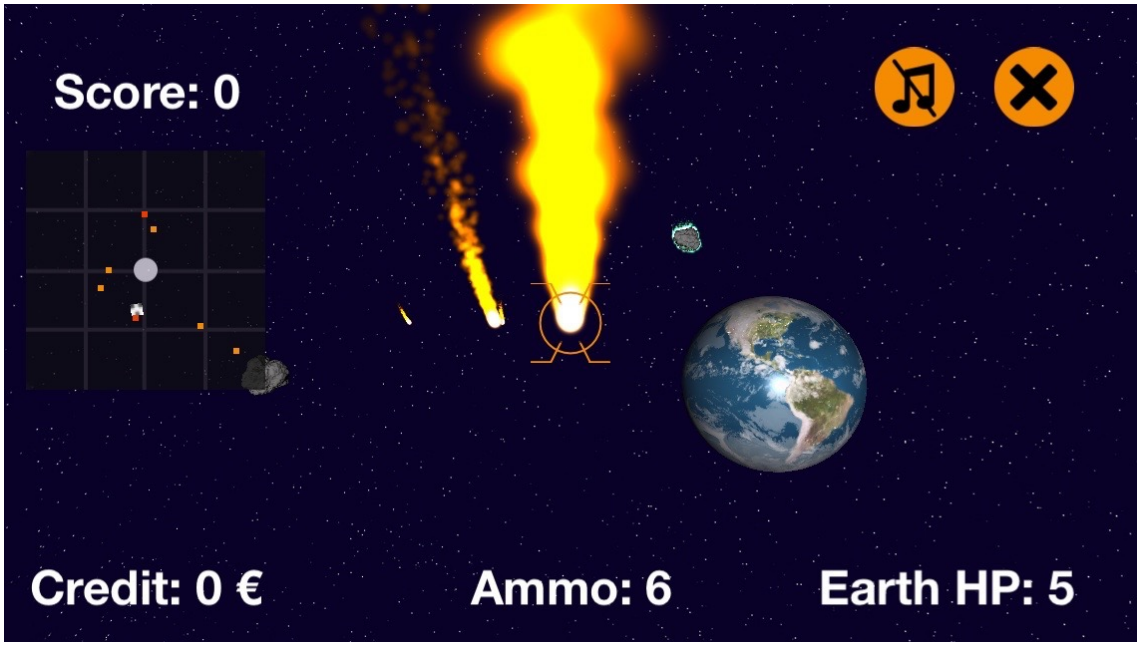

Image 5. Finished game screen.

SCNViews capability to have a SpriteKit scene as an overlay made it easy to show information to the player. As shown in image 5, this was used to display a radar that shows were objects are, game status information and buttons that can be used to quit or mute the game. The overlays were easy to setup but they also required to use SpriteKit in addition to SceneKit. However, the differences between SpriteKit and SceneKit are not very significant, so using it for a simple overlay did not require much additional learning.

5.2 The challenges faced

The project faced several small problems through its development. The biggest one of these was Swift's lack of maturity. The project has already gone through two large rewrites because of big Swift updates. Another thing caused by this was that the protocol oriented programming that Swift now supports, was not a choice when the development was started in Swift 1.2. The traditional object oriented way of developing with classes, was found cumbersome for avoiding code repetition and at the projects current state, repetition could not be completely avoided.

SceneKit had some undocumented features or glitches that were at times difficult to solve, because the low amount of overall documentation and discussion found for using the framework. Some of these were game breaking problems, such as the shopping screen that is shown to the player when a round in the main game is finished successfully, used to pause if there was nothing moving in the main SceneKit scene. Making the object for Earth stay in the scene and have it animated with SCNAction rotate forever eventually solved this problem, but it was still a nuisance to solve.

5.3 Future for the game

The game is still headed to eventually see release in the App Store after the issues raised by the user testing has been alleviated. Showing a view with detailed instructions before game starts will most likely be enough to fix the problems. The project might also be rewritten to the newer protocol oriented way of programming before this is done.

Swift has supported extending protocols with default behaviour from version 2.0. This makes it possible to abandon classes and instead only rely on structs, which are then made to implement their appropriate protocols (Kerber Erik, 2015). Using structs instead of classes avoids some problems, such as when classes are passes to around in the code, they will all change the shared classes state. However structs will always be copied so every usage of it will change only alter the used copy's state. Calling classes super methods when overriding a method are also a bit unreliable, as it is sometimes

hard to know at what point in the override it should be called: after or before the sub-classes own code?

Using only protocols in the project would require a complete rewrite and rethinking on how its architecture should work. It could, however, improve and simplify the game a lot.

6 Conclusion

The focus of this thesis was to study how SceneKit framework in unison with the new Swift programming language can be used to develop a functional game. This was all based on a project with a goal to make a game that is publishable to Apple's App Store. Requirement for this was that the project should pass all required guidelines in App Store Review Guidelines.

The thesis shows how to design a game project. The design was eventually split into two different games that both take place in space and share the common idea of saving object from meteors for as long as possible. The thesis also explained how the games graphics were made.

The main chapter covered how the key parts of the game were implemented, with simple examples taken from its source code. The idea was to show how the mains aspects of SceneKit actually work and can be done. It can also be concluded that SceneKit had more than enough features to handle the required game design.

The made choices were later reflected on: what was done, what problems were found when working on the project and what could be improved in the future. Based on the results it seems that it could be worth testing the newer protocol oriented way of programming now that Swift supports it. User tests that were done also show that although the game looks good, more instructions on how to play it are needed before it can be said to fulfil its objective completely.

In conclusion I believe that the objectives of the thesis were completed, though the game project still needs some improvements before its goals can be deemed reached. The thesis shows how SceneKit framework was used with Swift programming language to program an iOS game and it should help anyone looking to use these technologies in game development.

References

App Programming Guide for iOS (2016) [online]. Apple, 13 September 2016. URL: https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007072. Accessed 15 October 2016.

App Store Review Guidelines (2016) [online]. Apple, 2016. URL: https://developer.apple.com/app-store/review/guidelines/. Accessed 15 October 2016.

Cohen Peter (2014). Metal in iOS 8: Explained [online]. iMore, 27 June 2014. URL: http://www.imore.com/metal-ios-8-explained. Accessed 12 July 2016.

Create games, connect with your audience, and achieve success (2016) [online]. Unity, 2016. URL: https://unity3d.com/unity. Accessed 5 October 2016.

Common App Rejections (2016) [online]. Apple, 2016. URL: https://developer.apple.com/app-store/review/rejections/. Accessed 15 October 2016.

GameFromScratch.com (2015). GAMEDEV GLOSSARY: LIBRARY VS FRAMEWORK VS ENGINE [online]. GameFromScratch.com, 13 June 2015. URL: http://www.gamefromscratch.com/post/2015/06/13/GameDev-Glossary-Library-Vs-Framework-Vs-Engine.aspx. Accessed 12 July 2016.

iOS Data Storage Guidelines (2016) [online]. Apple, 2016. URL: https://developer.apple.com/icloud/documentation/data-storage/index.html. Accessed 15 October 2016.

iOS Human Interface Guidelines (2016) [online]. Apple, 2016. URL: https://developer.apple.com/ios/human-interface-guidelines/. Accessed 15 October 2016.

Jeremiah van Oosten (2012). Understanding Quaternions [online]. 3D Game Engine Programming, 25 June 2012. URL: http://www.3dgep.com/understanding-quaternions/. Accessed 1 October 2016.

Kerber Erik (2015). Introducing Protocol-Oriented Programming in Swift 2 [online]. raywenderlich.com, 25 June 2015. URL: https://www.raywenderlich.com/109156/introducing-protocol-oriented-programming-in-swift-2. Accessed 3 October 2016.

Myth Busting: Mobile Gaming Demographics (2015) [online]. RealityMine, 1 June 2015. URL: http://www.realitymine.com/myth-busting-mobile-gaming-demographics/. Accessed 15 October 2016.

SceneKit (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit. Accessed 2 October 2016.

SCNAction (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnaction. Accessed 2 October 2016.

SCNLight (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnlight. Accessed 9 October 2016.

SCNMaterial (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnmaterial. Accessed 9 October 2016.

SCNMaterialProperty (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnmaterialproperty. Accessed 9 October 2016.

SCNNode (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnnode. Accessed 9 October 2016.

SCNPhysicsContactDelegate (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnphysicscontactdelegate. Accessed 2 October 2016.

SCNSceneRendererDelegate (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnscenerendererdelegate. Accessed 1 October 2016.

SCNScene (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnscene. Accessed 9 October 2016.

SCNView (2016) [online]. Apple, 2016. URL: https://developer.apple.com/reference/scenekit/scnview. Accessed 9 October 2016.

Solt Paul (2015). Swift vs. Objective-C: 10 reasons the future favors Swift [online]. InfoWorld, 11 May 2015. URL: http://www.infoworld.com/article/2920333/mobile-development/swift-vs-objective-c-10-reasons-the-future-favors-swift.html. Accessed 12 July 2016.

Sparrow (2016) [online]. Gamua, 2016. URL: http://gamua.com/sparrow/. Accessed 5 October 2016.

Specifying the renderer for SpriteKit and SceneKit (2015) [online]. Apple, 8 October 2015. URL: https://developer.apple.com/library/ios/qa/qa1904/_index.html. Accessed 12 July 2016.

Tabini Marco (2013). Sprite Kit, GLKit, and Scene Kit: How Apple is shaping game development [online]. MacWorld, 13 November 2013. URL: http://www.macworld.com/article/2051345/sprite-kit-glkit-and-scene-kit-how-apple-is-shaping-game-development.html. Accessed 12 July 2016.

Trouble subclassing SCNScene (2015) [online]. Stackoverflow, 30 December 2015. URL: http://stackoverflow.com/questions/34534743/trouble-subclassing-scnscene/34536248#34536248. Accessed 14 July 2016.

Wenderlich Ray (2011). OpenGL Tutorial for iOS: OpenGL ES 2.0 [online]. raywenderlich.com, 25 May 2011. URL: https://www.raywenderlich.com/3664/opengl-tutorial-for-ios-opengl-es-2-0. Accessed 12 July 2016.

What is Cocos2d-x (2016) [online]. cocos2d-x.org, 2016. URL: http://www.cocos2d-x.org. Accessed 5 October 2016.

Zibreg Christian (2015). Unity 5 game engine launches with iOS Metal and 64-bit support and other improvements [online]. iDownloadBlog, 3 March 2015. URL:

http://www.idownloadblog.com/2015/03/03/unity-5-game-engine-launches-with-ios-metal-and-64-bit-support-and-other-improvements/. Accessed 12 July 2016.