VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

IOS GAME

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE   TOMÁŠ MIČIAK
AUTHOR

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HRA NA PLATFORMĚ IOS
IOS GAME

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE   TOMÁŠ MIČIAK
AUTHOR

VEDOUCÍ PRÁCE   Ing. MICHAEL ANGELOV
SUPERVISOR

BRNO 2012

Abstrakt

První dvě kapitoly pojednávají o různých možnostech vývoje her pro platformu iOS. Následující kapitoly se zabývají návrhem a implementací jednoduché závodní hry pro více hráčů. Důraz je přitom kladen na příjemné a intuitivní ovládání. Diskutovány a implementovány jsou víceré koncepty. V hře více hráčů zařízení komunikují skrze bluetooth rozhraní. V závěru je zhodnocena implementace, výsledky uživatelského průzkumu a dále jsou diskutovány možnosti dalšího vývoje.

Abstract

The first two chapters of this bachelor thesis discuss various options for the game development on the iOS platform. Succeeding chapters describe design and developments of a simple multiplayer racing game. This study is primarily focused on comfortable and intuitive game controls. It discusses and implements multiple concepts. In the multiplayer mode devices communicate via bluetooth network. The conclusion reviews the implementation results, evaluates the user survey and discusses possible improvements.

Klíčová slova

Apple , platforma iOS, Cocos2d, Box2D, mobilní zařízení, závodní hra, různá ovládáni, síť,

Keywords

Apple, iOS platform, Cocos2d, Box2D, mobile devices, racing game, various controls, network

Citace

iOS Game

Prohlášení

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Michaela Angelova. Uviedol som všetky publikácie a zdroje infromácií z ktorých som čerpal.

....
Tomáš Mičiak
May 16, 2012

Poděkování

Na tomto mieste by som rád poďakoval Ing. Michaelovi Angelovi, vedúcemu bakalárskej práce, za cenné rady, pomoc, ochotu a čas, ktorý mi počas tvorby práce venoval. Ďalej ďakujem všetkým ľuďom, ktorí sa podielali na testovaní a poskytli podnetné pripomienky.

Contents

Chapter 1

Introduction

Smartphones have come through major changes in the recent years. They are equipped with multitouch displays powered by high-performance RISC [1] processors. They usually do not contain hardware keyboard and the whole user interface is virtual manipulated via the multitouch display. This form of user interaction with the device is still relatively new. It is therefore essential that the user interface is as intuitive and user-friendly as possible. The another thing that changed is how we perceive them. In the past they were used almost exclusively by professionals. They were great communication and organizing tools, but nothing more. Nowadays their provide countless functions and have become stable parts of everyday activities of common people. They are still being used as a working tools, but also as an entertainment and multimedia stations. The main source of entertainment are games. In a contrary to computer games they are meant to be a momentary source of entertainment for masses of casual gamers. To satisfy this intention they are simple and fun to play. Their popularity is rapidly growing together with growing numbers of smartphones owners. This trend attracts more and more developers and the game industry is experiencing a renascence. Not only new games are being produced by also various game frameworks and tools. These technologies allow independent developers to create games equal to games created by large development studios.

This thesis is also focused on the indie[2] game development. In the first chapter it introduces iOS platform and describes native iOS technologies suitable for the game development. The next chapter mentions third-party game development frameworks sorted in multiple groups. The four most-often used are then described in more detail. The succeeding two chapters describe the process of designing, developing and testing a top-down 2D multiplayer racing game. The main goal of this game is implementing several intuitive controls. Its multiplayer is played via bluetooth network. Players are able to create and play their own level right in the application. The end of the thesis reviews the whole thesis and evaluates achieved results.

---

[1] Reduced Instruction Set Computer
[2] "Indie" is an abbreviation of the word independent

Chapter 2

iOS platform and game development

The first paragraphs of this chapter introduce reader to the iOS platform. The succeeding sections mention the game development possibilities with native iOS technologies. Last part discusses the game networking in more detail.

System

iOS is an Unix-like mobile operating system developed by Apple Inc. It is written in C and Objective-C and it shares many underlying technologies with Mac OS X, operating system for desktop computers and laptops. [9]

Objective-C

Objective-C is a dynamically typed programming language designed to bring object-oriented programming capabilities into the standard ANSI C language. These additions are mostly based on Smalltalk. [2] iOS version of Objective-C differs from Mac OS X version. The biggest difference is a missing Garbage collector in iOS version. Although it helps to improve performance, it also causes many memory-related programming errors. This drawback was reduced in iOS 5.0 by adding the Automatic Reference Counting system to help programmers reduce their memory-management code. [12]

Devices

iOS operating system is distributed only with Apple devices and does not support other manufacturers. There are three device families running iOS.

- iPhone (Multimedia smartphone)

- iPod touch (Multimedia player, its hardware is almost identical with iPhone's)

- iPad (table computer)

2.1 Interface development

User interface

User interface is built on the concept of direct manipulation and multi-touch technology. Interaction is based on multi-touch display where clicking with pointing device is replaced by finger gestures. Interface the part of iOS, where the difference between individual device families is most obvious. Universal applications have to deal with displays of various dimensions and resolutions. Every device supports changeable screen orientation. In iOS user at one moment always interacts with only one running application. [10]

UIKit Framework

This framework provides basic components for implementing any graphical, event-driven iOS application. [11] Its fundamental features are:

- Application management

- User interface management

- Support for handling touch and motion-based events

- Objects representing the standard system views and controls

- Integration with system and other applications

- Support for device-specific features (camera, accelerometer, etc..)

Core Graphics Framework

It is a rendering API used in iOS and OS X. It is based on Quartz a two-dimensional, resolution and device-independent drawing engine. [11]

OpenGL ES

The OpenGL ES is a cross-platform library written in the C language. It is an embedded systems version of desktop standard OpenGL. Its main function is drawing 2D and 3D graphics. [28] To Work with other iOS technologies, OpenGL ES is integrated into Core Animation by displaying its rendered contents in *CAEAGLLayer*. Apple devices implement both major standards OpenGL ES 1.1 and OpenGL ES 2.0. [13]

2.2 GameKit Framework

GameKit Framework provides game related networking services presented in form of Objective-C classes and interfaces. The main goal is to provide developers with means for developing multiplayer games. GameKit is available in iOS 3.0 and higher as a part of Cocoa Touch. [5]

Game Center

Game Center was introduced in iOS 4.1 as a social gaming service extending Game Kit framework. It is supposed to immerse players into playing games on iOS devices by connecting competitive multi-player games with a social network. Players can join by creating their Game Centre nickname called a *User Alias*. After joining the Game Center players are able to play with other players, add other players to their friend list, challenge and compete with added friends, view game leader boards, compare results and search for new games. Even single-player games can be connected to Game Center via achievements. Achievement is a badge, or an award for completing a certain part of a game, or for achieving some extraordinary score. Furthermore Game Kit offers in-game voice chat functionality without need to play via Game Center. [6]

2.3 Network connectivity

GameKit offers straightforward networking interface for peer-to-peer communication via bluetooth or local wireless network. Created network can at one time use maximum of sixteen devices. [7].

Sessions

Application on each device creates a session object that represents this application during communications with other devices. Application has to provide a delegate object that handles connection requests and data handler to receive data. Each session object has to create unique peer identification string and session identification string.
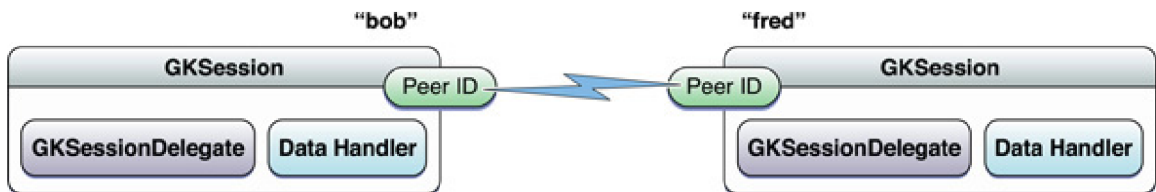

Figure 2.1: GKsession [8]

Discovering other peers

Searching for peers connected to the network is determined by session id[1], and a session mode. Session objects can be initializes with one of three session modes. These modes determine how will session act while discovering other peers, but they do not affect way of exchanging data. Peer, server and client acts as a peer able to send data to every connected device.

- Server advertises a service on the network

- Client searches for servers advertising themselves on the network

---

[1]Session Identifier is a unique string identifying the service. Only devices running applications with the same session id can be connected.

- Peer advertises as a server and searches as a client simultaneously. This causes longer delay before discovering other devices.

Peer-to-peer network supports only up to four connected devices, while client-server supports up to sixteen connections. Furthermore if is device discovery and connection done through provided *GKPeerPicker* interface, number of connections is restricted down to two for both architectures.

Managing connections to other peers

As was already mentioned session objects handle connections. To work with sessions *GameKit* framework provides *GKSession Delegate* interface with one required method `session:peer:didChangeState:`. It is invoked every time any of the peers in the session changes its state. Peers state changes when:

- After becoming discoverable by the session (Peer initializes his session and starts propagating himself)

- While connecting to the session (Peer tries to connect to the session)

- After the connection has been establishing

- After disconnecting from the session

- After becoming unavailable for the session

These five states are defined in *GKPeerConnectionState* enumeration. However accepting or declining connection is advised to manage by implementing function `session:didReceiveConnectionRequestFromPeer:`.

Exchanging data

After the connection process, peers are able to exchange data. Two delegate functions are available for sending application data either to a subset or to all connected peers. Application may send any possible data and objects types. The only constraint is that, they have to be encoded to a *NSData object*. For best performance it is recommended to keep size of the data objects under 1000 bytes. Messages larger than 1000 bytes may need to be split into smaller chunks and reassembled at the destination, resulting in additional overhead and thus latency.

Send data methods modes

Application data can be send in one of two modes, *GKSendDataReliable* and *GKSendDataUnreliable*, corresponding to TCP or UDP transmission.

Chapter 3

Third-party frameworks and tools

The beginning of this chapter discusses the benefits of using a third-party technology in the game development. Four major frameworks are described in detail. At the end of the chapter is a list of the game development tools. Each tool is briefly described.

Introduction

Most of the games for iOS platform are implemented with help of some third-party technologies. Creating frameworks is a common practice. It broadens functionality of default APIs and brings some missing features. In case of iOS programming interfaces provided by Apple are not very suitable for quick and easy game development. To compensate this drawback, third-party developers created large number of various frameworks. These can be divided according to multiple categories, such as:

- **Free frameworks:** Cocos2d, Sparrow, Chipmunk, Box2d

- **Payed frameworks:** Unity, Unigine, Sio2, Corona, Chipmunk Pro

- **Objective-C frameworks:** Cocos2d, Sparrow

- **Cross-platform frameworks:** Chipmunk, Unigine, Box2d, Unity, Corona

Two most widely used frameworks are Cocos2d and Sparrow. Both are open-source graphical Objective-C APIs based on OpenGL ES. They offer very similar functionality for creating and structuring graphical applications. Each has large a community of developers, who actively contribute by creating extension and by adding missing features. Their key features are:

- Management of hierarchically displayed graphical objects

- Integration of Cocoa touch event system

- Various methods for rendering, sprite manipulation and animation

3.1   Sparrow framework

Sparrow was inspired by Flash programming language. Its API is built to be very similar, to ease the transition for Flash games programmers. [17]

Display Objects

Every displayable object in Sparrow is a subclass of SPDisplayObject. SPDisplayObject is an abstract class therefore it cannot be used directly. However it provides many methods and attributes that are utilized by all other display objects. Another special abstract class is SPDisplayObjectContainer. This class acts as a container for other display objects. It allows objects to be organized into a logical system called The Display Tree.
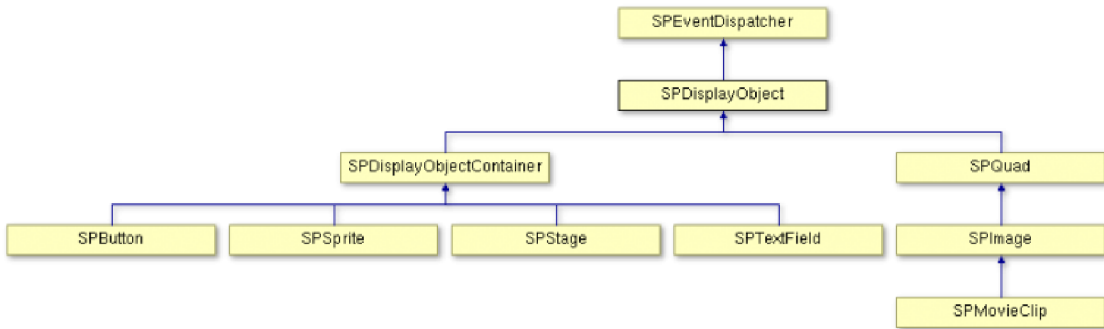

Figure 3.1: Display objects class hierarchy [18]

The Display Tree

Since the SPDisplayObjectContainer is abstract class, organizing of objects is usually done by using SPSprite objects. Every display object container has its own coordinate system, which may be different from its parents. Figure 3.2 shows coordinate system on two layers. The red pin represents root of the child's coordinate system as well as point relative to parent coordinate system.
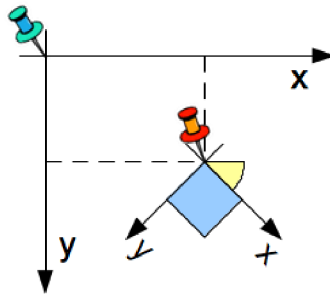

Figure 3.2: Sparrow coordinate system [19]

Event handling

Sparrow's event system is tightly coupled with the display tree. SPDisplayObject and thus every other content displaying object inherits from SPEventDispatcher. Therefore each of these objects is prepared to handle events. There is special type of events called bubbling events. They can be created in a leaf node of the display tree. Then they are handed from child to parent until reaching the root node of the tree. Any of display objects along

this route can listen to bubbling event and stop it from advancing further. Mostly used type of bubbling events are touch events. Each touch event has four phases to help recognize the type of a touch. These phases are SPTouchPhaseBegan, SPTouchPhaseMoved, SPTouchPhaseStationary and SPTouchPhaseEnded.

Animations

There are two ground types of animations. First are firmly defined animations, such as fading, moving, scaling. Second are animations dynamically affected by user interaction. These dynamic animations are available to every display object. To use it, object has to implement a listener method for SP_EVENT_TYPE_ENTER_FRAME events, which is then called every time a new frame is drawn.

3.2 Cocos2d framework

Cocos2d, as its name suggest is a framework designed for creating 2D games, however there is already an early release 3D version. [26]

Graphical objects hierarchy

Similarly to Sparrow, displayable objects in cocos2d are also organized into a tree structure. Most important classes for creating this tree structure are *CCScene*, *CCLayer* and *CCSprite*, each is a subclass of *CCNode*. *CCNode* objects can be modified either by manually changing its properties, or by applying *CCAction* method. However drawing the object is done in its *-(void)draw* method.

Scenes

Every Cocos2d application is composed of a set of *CCScene* objects. Scene takes up the whole screen and serves as a root in graphical nodes hierarchy. Although application usually contains many scenes only one of them is active at the moment. They may be connected by *CCTransitionScene* scenes which define scene-swapping logic. These transition scenes provide various animations, such as fade out, fade in, slide from the side, etc.

Director

CCDirector object is the main component responsible for switching back and forth between scenes. It is a shared (singleton) object. It knows which scene is actually active and it handles a stack of scenes to allow scene calls (pausing a scene and putting it to hold, while another enters, and them returning the former one).

Layers

A CCLayer object has a size of the whole drawable area and knows how to draw itself. It can be semi transparent (having holes and/or partial transparency in some/all places), allowing to see layers behind it. Layers give scenes appearance and behavior since they implement event handlers. Events are propagated to Layers (from front to back) until some layer catches the event and accepts it. Layers can accept any CCNode subclass object as a child object. Only exceptions are *CCScene* objects and objects from its subclasses.
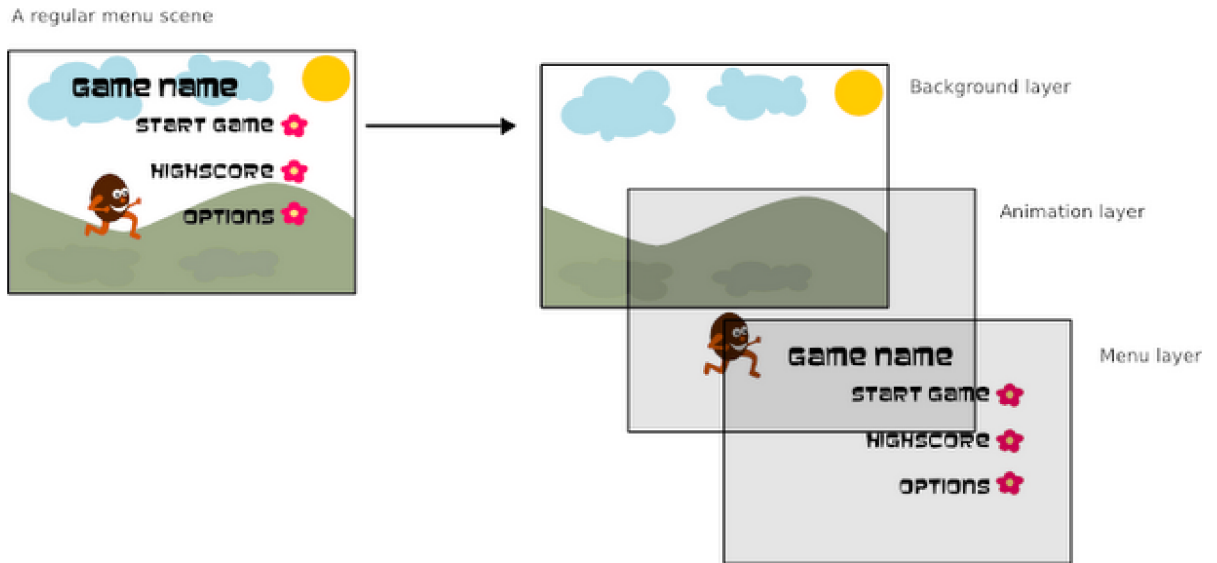
Figure 3.3: Layers system in Cocos2D [27]

Sprites

CCSprite objects represents an image, that can be moved, rotated, scaled, animated, etc. Sprites can have other sprites as children. When parent is transformed, all its children are transformed as well.

Actions

Actions are usually modifying some of objects attributes (opacity, position, rotation, scale, etc.) to create animation. *CCAction* class has two subclasses. *CCIntervalAction* actions are being invoked for a certain period of time, while *CCInstantAction* actions are invoked only once. Each *CCAction* object has several properties, which specify how the action modifies *CCNode* object. All actions can be paused/resumed by using the CCActionManager. There are two basic categories of *CCIntervalAction*:

- Relative actions (their names end with suffix By)

- Absolute actions (their names end with suffix To)

Each relative action and some absolute actions can be executed in a reverse direction. Multiple actions can be composed together into one of the four types of action compositions:

- Sequence is a list of actions executed in the order they were defined.

- Spawn is a list of actions executed at the same time.

- Repeat composition differs from the previous two. It is applicable only to one action which is repeated for a finite number of times.

- Repeat forever is the same composition as the Repeat composition except for unlimited repeating of the action.

Effect actions

These actions are different from others since they do not modify normal object properties. They render any CCNode object into a grid, which can be modified by moving its vertices. There are two types of grids, tiled and non-tiled grids. The difference is that the tiled grids are composed of individual tiles while non-tiled grids are composed of vertices. Although the grids has 2 dimensions (rows and columns) each vertex has 3 dimensions, which allows execution not only 2D, but also 3D effects.

Touch events handling

Cocos2d offers two types of handling touch events by implementing given delegate methods. Only one of these can be used at the time: Standard touch delegate optionally implements four methods handling beginning, movement, canceling and end of each touch event. These events are propagated from the Cocoa touch environment along with set of touch points represented by *UITouch* objects. Targeted touch delegate implements almost the same methods. Only difference is, that they are called form each touch separately, thus targeted touch delegate is not suitable for handling multitouch.

3.3 Physics engines

Reason to use any of the physics engines is mainly collision detection and simulation of the game objects. There are only two largely used physics engines, Chipmunk and Box2D. Both are open-source two-dimensional rigid body libraries.

3.3.1 Collision detection in general

Collision detection is a substantial part of every dynamic game. There are various ways how to detect collision, some are very simple and computationally lightweight. However most of the games require pixel-point precise algorithms which can be computationally very demanding. Since collision detection computing time is crucial to the overall game performance usually are the two approaches combined. [22]

Boundary sphere check

It is by far the easiest and fastest method. In this method it is important that every graphic object is situated roughly in the center of its containing sprite. First step is to measure distance of the objects. Each object is traced out in an imaginary circle. If circles of two objects collide it is possible that collision has happened. Although as was already mentioned precision of this method depends on the shapes of the objects. Therefore it is ideal to combine this method with some more advanced algorithm. This way the performance of collision detection is improved, because not colliding circles are quickly located and omitted from further evaluation.

Bounding box check

It is similar to the previous method. Objects are traced out in an imaginary rectangles rotated according to the world's axes. For this method to stay easy and fast rectangles do not support rotation. This is a problem if game objects need rotate freely. Possible

solutions are one bounding box large to covert object in any rotation, or set of boding boxes, each mapped to certain angle.

Pixel-perfect detection

Each sprite is rendered into a pixel mask, a two-dimensional boolean or bit array. It records whether pixel can or can not collide, in other words whether it is or it is not opaque. Whether a pixel is or is not opaque is determine by alpha threshold. Collision detection algorithm itself is then composed from cycle-checking all pixels from two masks for collision. Optimization of this method is firstly defining which parts of masks are overlapping and then checking only those parts.

Physics engine collision detection

Both Chipmunk and Box2D use similar collision detection techniques. To increase the computation speed they break the whole process down into several stages implemented by various algorithms. These are described in the next sections.

3.3.2 Chipmunk

Chipmunk is written in C, but there is also a commercial Objective-C version. Chipmunk defines own data types and mathematical operations with these types. Objects simulated in chipmunk are called Rigid bodies. They contain most of the object's physical properties. However they do not define the actual shape of the object. Body's shape is defined by attaching set of collision shape objects. Collision shapes have two main purposes. Firstly they define the actual shape of the body used in collision detection. Secondly they define additional physical properties such as surface friction and elasticity. Multiple bodies can be connected through series of joints and constraints. After creating a body, attaching collision shapes to it and defining its joints and constraints, body is usually inserted into a space. Spaces are simulation containers for rigid bodies. Only bodies in the same space may interact with each other. All objects contained in a space are automatically simulated. Thus by not adding body to a space one can create so called *Rogue bodies*. Since simulation doesn't affect them they movement can by controlled directly. Special type of *Rogue bodies* are *Static bodies*. Each space has one dedicated *Static body*. Attaching shapes to static bodies defines level structure.. [20]

Body properties

Each Body has several properties defining its behavior in the simulation world:
Mass affects how forces and impulses converts to velocities
Moment of inertia represents rotational mass of the body
Position of the center of gravity of the body
Linear velocity of the center of gravity
Force affecting the center of gravity
Angular rotation of the body in radians
Angular velocity of the body in radians per second
Torque affects the center of gravity

Forces and impulses

Chipmunk bodies can be affected either by applying impulses or forces. Applied impulse is transformed into an instant change in body's velocities, while applied forces affect velocities continuously. How much will force or impulse affect the body depends on body's mass and moment of inertia. Applying of linear or angular force or impulse is done by calling corresponding method. While applying forces and impulses programmer may specify point where will these be applied. This point is specified as an offset from the center of body's gravity. This however cannot be done for torque, which is always applied to the center of gravity.

Collision detection

As was already mentioned, collision detection is based on collision shapes. There are three types of collision shapes, circles, convex polygons and line segments. Line segments are meant to be used with static bodies to define level environment. Therefore even though they can be actually added to a non-static body, they do not generate any collisions with other line segments.

Spatial Indexing   Purpose of this algorithm is to figure out which objects should be be checked for collisions. Currently Chipmunk supports two spatial indexes, an axis-aligned bounding box tree and a spatial hash. These spatial indexes are able to quickly identify which shapes are near each other and should be checked for a collision.

Collision Filtering   After the spatial index figures out which pairs of shapes are likely to be near each other, it passes them back to the space to perform some additional filtering on the pairs. Before doing anything else, Chipmunk performs a few quick tests to check if shapes should collide. One of them is mentioned bounding box test. Other two are:
Layer Test: Space in chipmunk may be composed of multiple layers. Each shape can occupy only one of these layers and collide only with shapes in the same layer.
Group Test: Each shape can belong to multiple groups and similarly to layers, shapes collide only with shapes in the same non-zero group.

Primitive Shape to Shape collision detection   This is the most expensive from the collision detection tests. It is based on object's geometry. There is a separate algorithm for each collision shape type. As was already mentioned, circular detection is fast, however polygon collisions are demanding and the time consumption increase depending on the number of vertexes. Therefore simpler shape results in fewer collision points and thus in faster detection.

3.3.3   Box2D

Box2D is written in C++ and it is very similar to Chipmunk, which was actually inspired by Box2D. It consist of three modules, *Collision Module*, *Dynamics Module* and *Common Module* It is ideal for simulating fast moving objects due to its specialized algorithm preventing fast moving objects from tunneling. [16] Tunneling is explained further in this section.

Unit system

Box2d works with floating point numbers, defines own data types and uses proprietary unit system. Units of this system are meters, kilograms, seconds and radians. This system is tuned to perform well with dynamic objects between 0.1 and 10 meters and static objects up to 50 meters. Since Box2D is not working with pixels programmer has to define conversion ratio between screen pixels and meters.

Integrator

It is a discrete computational algorithm for solving physics equations. It is extended by constraint solver. In each computational step constraint solver solves all the constraints in the simulation. While single constraint can be solved perfectly it affects other constraints. For this reason Box2D solves constraints multiple times in one step. Furthermore constraint solver is composed from two phases, a velocity phase and position phase each with separate user-definable iterations count.

Collision shapes

- Circular shapes cannot be hollow

- Polygon shapes are always convex and as well as circular shapes, they are never hollow. They are created by defining vertices in counter clockwise winding. Each shape has a radius property, which creates slim border around the shape and helps with collision detection. Furthermore it prevents against tunneling by keeping polygons slightly separated

- Edge shapes are Box2D versions of Chipmunk's line segments.

- Chain shapes are used to compose edge shapes into complex structures. Chain shapes provides two sided collisions and are able to create loops, however self-intersection is not supported.

Fixtures

Fixtures have two functions. They bind collision shapes to the body and define additional body properties connected to collision shapes.
Density is used to compute mass of the body.
Friction between two fixtures reducing velocities of their bodies. Friction can has values from the interval ¡0, 1¿, when 0 turns friction off and 1 is maximum friction. Overall friction of the contact is computed by geometric mean of the two touching fixtures.
Restitution defines how elastic the fixture is. Since all fixtures are rigid on the time of a impact the force affecting the body is either transformed or consumed by the impact. Ratio of the two is given by restitution of both fixtures.

Tunneling

Is a effect when due to discrete nature of the simulation in Box2D, fast moving objects may in one step of the computation jump or tunnel through other objects. This unwanted behavior is in Box2D prevented by implementing two additional algorithms. First time of

the impact for each body pair is computed. If it is smaller than duration of a time step, second algorithm resolves impact with smaller steps.

Damping

It is a body property similar to friction. However unlike friction, damping reduces an object's velocity at any time the object moves. There two types of damping, linear and angular damping. Each affects corresponding velocity.

Sensor collisions

Any collision shape in Box2D can be marked as a sensor shape. Sensor shapes do not produce standard collision, therefore they are able to overlap with other collision shapes. However they still produce collision callbacks which are send to a instance of a so called *b2contactListener*. By subclassing this class programmer is able to implement own collision behavior.

3.4 Tools

3.4.1 Zwoptex

Zwoptex is a freeware sprite sheet creator. It contains algorithms to trim, sort, rotate and layout added sprites or to reduce number of sprite's colors. The output of this software is one image file with all the sprites and depending on framework one XML or property list metadata file. It supports png and PowerVR image formats and most widely used game frameworks such as Cocos2d, Sparrow, Corona, Unity, Flash. User is also able to define own metadata file formats. [31]

3.4.2 Tiled

Tiled is a free multi-platform general purpose tile map editor. [29]

Tile mapping   is a technique which composes larger graphic objects by re-using a number of smaller ones. Main goal of this process is to increase memory efficiency and rendering performance. Map tiling combined with scrolling enables to create large game environments. [23] There are three types of tile maps, each type is defined by perspective in which it is composed.

- Orthogonal maps

- Isometric maps

- Hexagonal maps

Tiled supports orthogonal and isometric maps. Important function is ability to create maps in multiple layers, this allows users to manipulate and rearrange whole tile-composed objects. Maps are exported in XML-based format called *Translation Memory eXchange*, file suffix is .TMX. This format is supported by Java and Android *lib tiled-java* library, Cocos2D, Corona, Unity, Microsoft XNA/.NET, AndEngine, melonJS HTML5 game engine and many others.

3.4.3 Cocos Builder

Cocos builder is a free WYSIWYG editor for Cocos2d. It can be used to create *CCNode* graphs such as menus, levels and particle systems User is able to use almost all *CCNode* subclasses and access their properties. Also custom subclasses of *CCNode* are supported. However buttons created with Cocos Builder are able to use only selector callbacks and not blocks. Created node graphs can be loaded into any Cocos2d project with *CCBReader* class, shipped with this tool. [30]

3.4.4 PhysicsEditor

PhysicsEditor is a tool for creating physics engine's collision shapes definitions from sprite images. Although it is available for free, export without the payed license is restricted to only 10 shapes per file. User is able to manually define shape of the sprite by adding polygonal and circular shapes. Since this process can be really time consuming, PhysicsEditor provide automatic shape tracer. User can set alpha threshold to omit parts of the image, and precision tolerance of the process. Tolerance setting affects number of used line segments. If the tolerance is set to 1.0 tracing of the shape is done with pixel-perfect precision. Depending on chosen physics engine, user can define all specific rigid body and collision shape parameters. A custom file with all the properties is then exported. Shapes from this file can be loaded to any project by loader classes shipped with PhysicsEditor. This loader classes are available for AndEngine, Box2D, Chipmunk and Objective-Chipmunk. For other frameworks, such as Corona or Flash, PhysicsEditor generate source files, which can be directly added to a project. [4]

Chapter 4

Design

This chapter describes the design phase of the development process. In the first section it defines the game concept. Succeeding section then explains the main benefits of a physics engine and describes model of the car. Then the four types of controls are discussed and their design depicted. Network architecture and communication protocol are presented and figured by graphs. The last section describes the concept and design of the level creator tool.

4.1 Concept

Concept of the game is rather simple. It is meant to be a classic top-down 2D racing game. It has two modes, single-player and multiplayer. Multiplayer mode is played through via network created by the game. This unusual concept allows players to compete against each other without any internet connection. Another similarly unique feature among iOS games is the ability to create and play own levels.

Single-player

In single-player player competes against the time. He can choose any of the existing levels, or create a new one. After finishing the race, a screen with his overall time and the time for the best lap is presented. His best times are also compared to previous best times.

Multiplayer

In multiplayer players compete against each other. In most multiplayer racing games, the winner becomes the player who crosses the finish line first. However since the game supports multiplayer for up-to eight people, cars do not start from the same position. Cars are widely spread to avoid mass collision during the start of the race. If winner was determined by crossing the finish line this would cause an advantage for players closer to the start line. Therefore game is tracking time for each player and after all players have finished the race winner becomes the player with the best overall time. After finishing the race player is presented with his best times. But on the contrary to single-player, multiplayer supports only predefined levels and these do not save the previous best times. This is due Apple policy prohibiting writing into application bundle.

As was already mentioned in the introduction, game controls can really affect the user experience. Therefore the design of the game is focused at multiple pleasant and responsive

car controls. Together with this trait, the three things that make this game distinguishable from other games available for iOS devices are:

- Various sets of controls

- Local multiplayer via bluetooth

- Ability to create and play new levels right in the game

4.1.1   Game interface design

All game information required by player should be placed in a manner, that it can be easily and intuitively spotted without distracting the gameplay. Figure 4.1 shows placement of interface elements in the game. Game controls are omitted due to each type has different composition and placement. Game controls are discussed in the section 4.3 on page 20. Game information contains current lap time and count and time from the start of the race. Mini-map shows the whole track and current positions of the players.
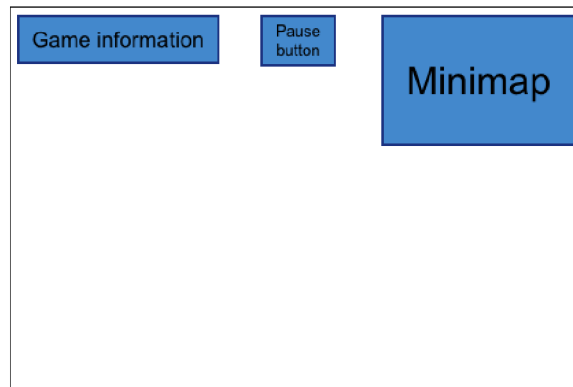
Figure 4.1: Positioning of various elements in the scene of the game

4.2   Physics

One of the reasons why a racing game is entertaining is its physics model. Generally speaking model in a racing game could be either realistic, or non-realistic. More often the non-realistic model is implemented. For two reason I chose to implement non-realistic model as well. Firstly implementation of a realistic model is very difficult and would exceed extent of this thesis. And secondly the game created in this thesis is a top-down 2D game running on a mobile device. And no mobile device has any means to comfortably control realistically moving car.

Physics engine

Although collision and movements of the car can be solved without the physics engine, without it, collision would not affect the car's movement and the game would lost the main reason why it is entertaining. After studying and trying both engines mentioned in section 3.3 on page 11, I found Box2D to be more fitting for this game. Mainly because of the linear and angular damping. Chipmunk do not provide any similar functionality. But

also because Box2D contains tunneling prevention and it's simulation of a impact seemed more precise.

Sensor collision detection

Collision detection between cars or between a car and a track border is done by standard Box2D collisions without any additional computing. However sensor collisions are used to count laps for each car. There are two sensors rectangles placed on the finish line, one from each side. For car to enter a new lap all fixtures of a car has to move firstly through first and then through second sensor line. This methods prevents from accidents which would cause multiple increments of the lap counter. However player can still cheat by going forwards and backwards through the finish line. This issue can be solve by adding additional checkpoints through the level.

4.2.1 Model

It was already mentioned that car's model is not realistic. Nevertheless movement of the car still has to seem realistic. Physics model of the car is implemented by implementing these rules:

- Acceleration and deceleration is done by continuous applying of linear impulses

- Car has set different maximum speeds for forwards and backwards movement

- Acceleration is dynamic, in higher speeds is acceleration faster and vice-versa

- Deceleration is also dynamic, in higher speeds car decelerates faster than in the lower speeds

- Steering is done by applying angular impulse in combination with torque

- Steering causes car to slow down

4.3 Game controls

Design of the various game controls was one of the main goals of this thesis. Although controls are not the reason why people choose a game, they can largely affect user experience. They should be designed as seamless part of the game. Player should be able to use them intuitively or adapt to them without any major problems. When it comes to racing games, the most important characteristic of game controls is the precision. In these games, controls are tightly bound to the physics model of the car. Importance of the car's model was described in the previous section. Output of every type of controls is adapted to the Box2D model of the car. It is composed of the two float values representing *linear* and *angular impulse*.

4.3.1 Virtual analog stick

Analog stick has always been very popular game controller. It was introduced as a part of a hardware game controller. It is used to control movement, rotation of the object or the game camera. It is used because of its precision and easy handling. User is able to control it by a single finger, usually by thumb. This is really convenient in games with more complicated controls system. For example in 3D games another analog stick is added for controlling the camera rotation. Virtual stick copies this concept and is composed of two parts, stick's background determining the range of motion and stick's nub which represents the actual stick.
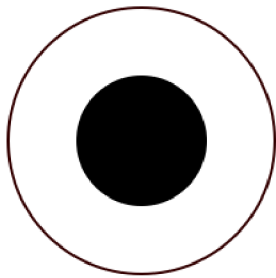
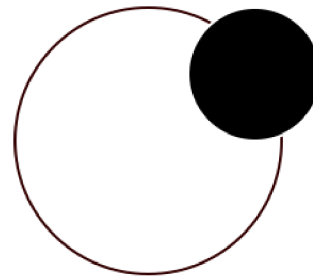

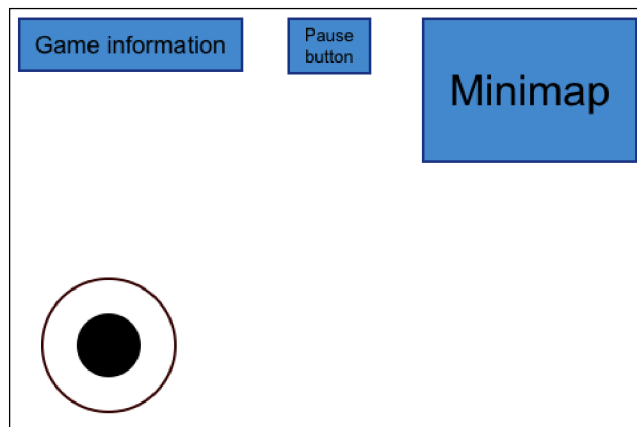Figure 4.2: Analog stick   Figure 4.3: Analog stick with moved nub

Figure 4.4: Virtual analog stick

In this game one stick is enough. It is able to control car's velocity as well as car's rotation. Sticks coordinates and the center of coordinate system always create right-angled triangle. Therefore stick's distance from the center and angle can be deduced by using Pythagorean theorem and Pythagorean trigonometric identity. Position of the stick represents the desired car's direction and rotation. Linear and angular impulses are computed to gradually adjust car's velocity and rotation according to the position of the stick. However rotation is adjusted according to the car's current rotation as well. Without position of the car it would be impossible to determine desired direction and magnitude of the rotation.
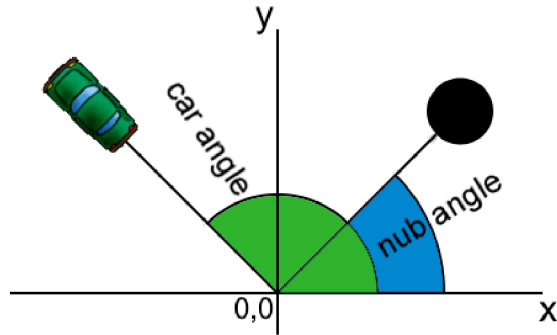
Figure 4.5: Car and the nub in the coordinate system

Classic analog stick supports only forward movement, when object always rotate to face the movement direction. This may lead to car getting stuck after colliding with the track border. To avoid this problem this stick implementation allows car to move backwards. This feature can be used also as a form of a brake. To use this function user has to direct the stick in the opposite direction of the car. However since car is primarily supposed to go forwards and this function is meant to be used only if the car gets stuck, it has reserved only 90-degrees wide angle 4.6.
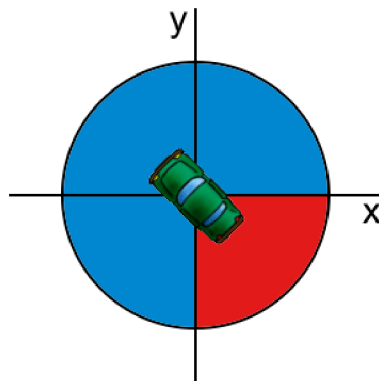

Figure 4.6: Cars reverse gear range

Algorithm sequence

1. Convert stick's position identified by y and x coordinates to distance from center and rotation. Distance is equal to $sqrt(x^2+y^2)$ and rotation is equal to $arcsin(y/distance)$.

2. Angular impulse is calculated from current angles of the car and the stick.

3. Linear impulse is calculated from stick's distance from the center of the coordinate system.

4.3.2 Virtual arrow keys

Arrow keys are most widely used way of controlling racing games on desktop computers. Virtual arrow keys, if properly designed can be very usable as well. In the next two paragraphs are described the main design issues that have to be solved.

Hand position is very different while using mobile device, than it is on the keyboard. Players usually hold the device with both hands, so they are able to control the game only with thumbs. Therefore a good way of improving the comfort is to divide four keys into two groups of two. Each group is placed in the area accessible by either of thumbs.

Pressing key on a device screen differs from pressing the actual physical key by lack of the sense of touch. Since player needs some sort of feedback, it is reasonable to replace this by distinguishing looks of the key while it is pressed. Another possibility is using the vibration mechanism. However in a game which uses more control keys, or in a game where control keys are used frequently this method may become useless.

Gas and brake pedals

This concept is most similar to actual keyboard keys. There are two pairs of keys, two for steering and two for controlling velocity. Each key may be pressed and held, but when the other key from pair is pressed the former is automatically released.
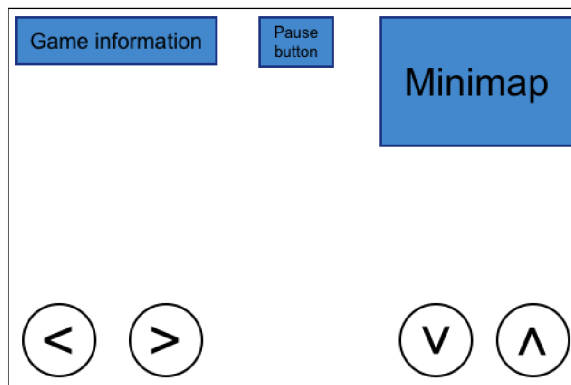

Figure 4.7: Arrow keys with gas and brake pedals

Velocity slider

Steering is the same as in the previous concept, however gas and brake pedals are replaced by an virtual two-directional analog slider. Initial position of the slider is in the centre of its bounding rectangle. By pushing the slider upwards the car speeds up and vice-versa. If player releases the touch, slider, depending on the speed of the car, returns to its initial position.
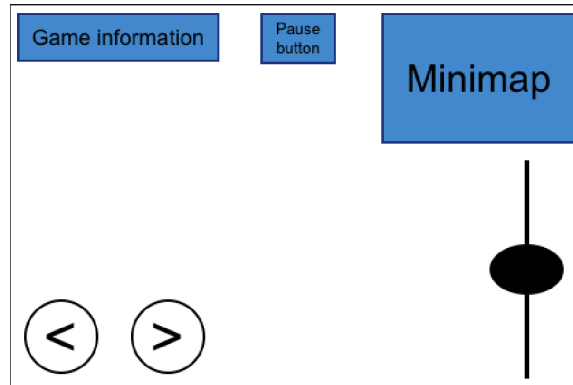

Figure 4.8: Arrow keys with velocity slider

4.3.3 Accelerometer

Every iOS device contains a triaxial accelerometer, an electromechanical device that is able to measure acceleration forces along three orthogonal axes. [3] These forces can be either static, for example constant force of gravity, or they can be dynamic, caused by the movement of the device.
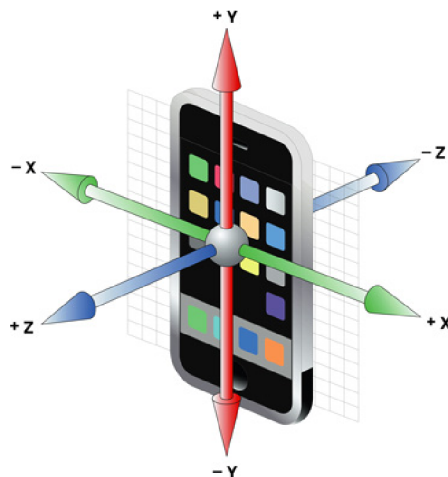

Figure 4.9: Accelerometer axes [14]

Accelerometer is in iOS devices accessible through the *UIKit* framework. It defines one delegate method, which receives *UIAcceleration* object holding information about the position of the device. This object has four properties, one float value for each of the axes and a timestamp. [15]

It is possible to control car only by accelerometer without any additional controls. One axis would control steering and another would control velocity. However this way of controlling is not suitable for iOS mobile devices. It is because range of motions can be either large, then controlling may lead to excessive motions which with combination with low viewing angles cause user to lost orientation. On the other hand if the range of motions is small, controlling becomes inaccurate. For this reason the best solution is to use one axis for steering the car, while use additional on-screen controls for car's velocity. This is the method implemented in the game. Velocity is controlled by the same gas and brake pedals which are used with arrow keys controls. Only difference is that these pedals are not placed next to each other, but on the opposite sides of the screen. For car to go straight, player has to hold the device in the initial position. The more the device is tilted, the faster the car is turning.
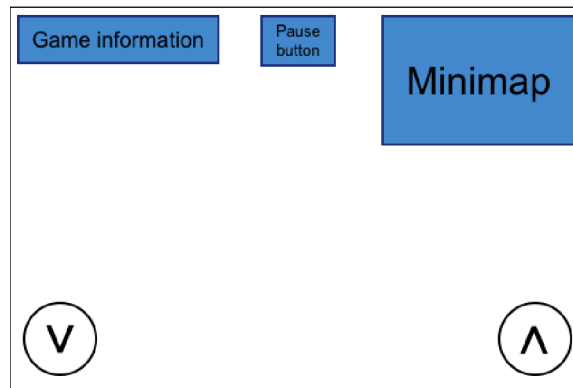

Figure 4.10: Accelerometer controls

To allow user to hold the device in a comfortable position, calibrating function is available. This function creates a new initial position for the device and all other positions are calculated accordingly to this new position.

4.4 Network communication

4.4.1 Architecture

Every game network architecture is based on one of the two opposite concepts, a peer-to-peer or a client-server. Advantages as well as disadvantages of both are described in the following paragraphs.

Peer-to-peer

In peer-to-peer architecture each device has its own copy of the simulation world. Devices send each other the user inputs, in this case the two float values. This solution consumes only a little network bandwidth. However without any additional synchronization any network lag causes inconsistency in simulations between individual devices.
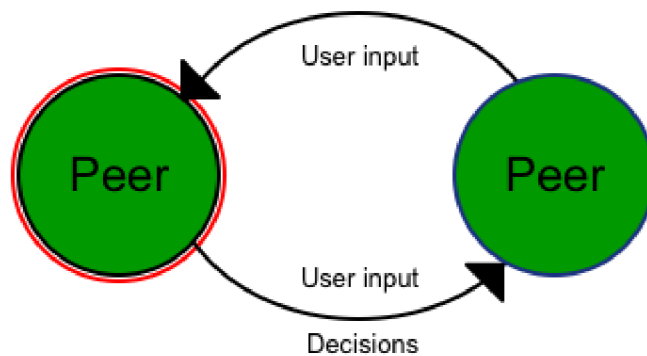
Figure 4.11: Peer-to-peer architecture

Client-server

Client-server architecture do not suffer from aforementioned problem. On the other hand server has to send a message after each change in the simulation world. Depending on the simulation this could cause increased server and network load.
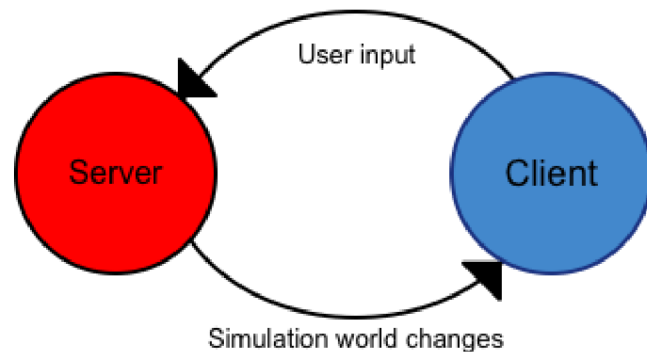
Figure 4.12: Client-server architecture

Since simulation in the game is not that extensive, I have chosen to implement a client-server architecture. One of the players creates and hosts the game acting as a server and others join the created game as clients.

4.4.2   Communication protocol

Packets in the communication protocol of the game are either message packets or game-update packets. To achieve better network performance, game-update packets, are transmitted via UDP protocol. It is no problem if any of them gets lost, because they are being sent constantly one after another. On the other hand it is critical that message packets are delivered. They are therefore sent via TCP. They inform about game-related changes e.g. instruction to start or load the game. There is no need for handling session state changes by sending message packets, because this is already implemented in *GameKit* framework. How to handle session changes and how to send data via mentioned protocols is described in section 2.3.

4.4.3   Communication phases

Communication process can be divided into three parts. They are described in the following paragraphs and displayed on pages 27 and 28.

Starting the game

Connection is always initiated by the client. After accepting clients connection, server sends information about peers order and the selected level. While waiting in the game lobby server sends ping packets to all connected peers. It is done to prevent them from becoming inactive and therefore disconnecting from the session. When player acting as a server hits the play game button server sends out instruction to load the game. After then when all clients have game successfully loaded server sends out another instruction to start the game. The start phase is ended by this message.

In-game communication

During game devices exchange game data packets. Clients send processed user input and server sends the simulation updates. Both are able to freely pause and resume the game at any time.

Ending the game

Even after the player acting as a server finishes the game, simulation and packet exchange still runs to enable other players to finish as well. After finishing the race players are redirected to the main menu and connection is terminated. Safe disconnection is done using *GameKit* API without sending any messages.
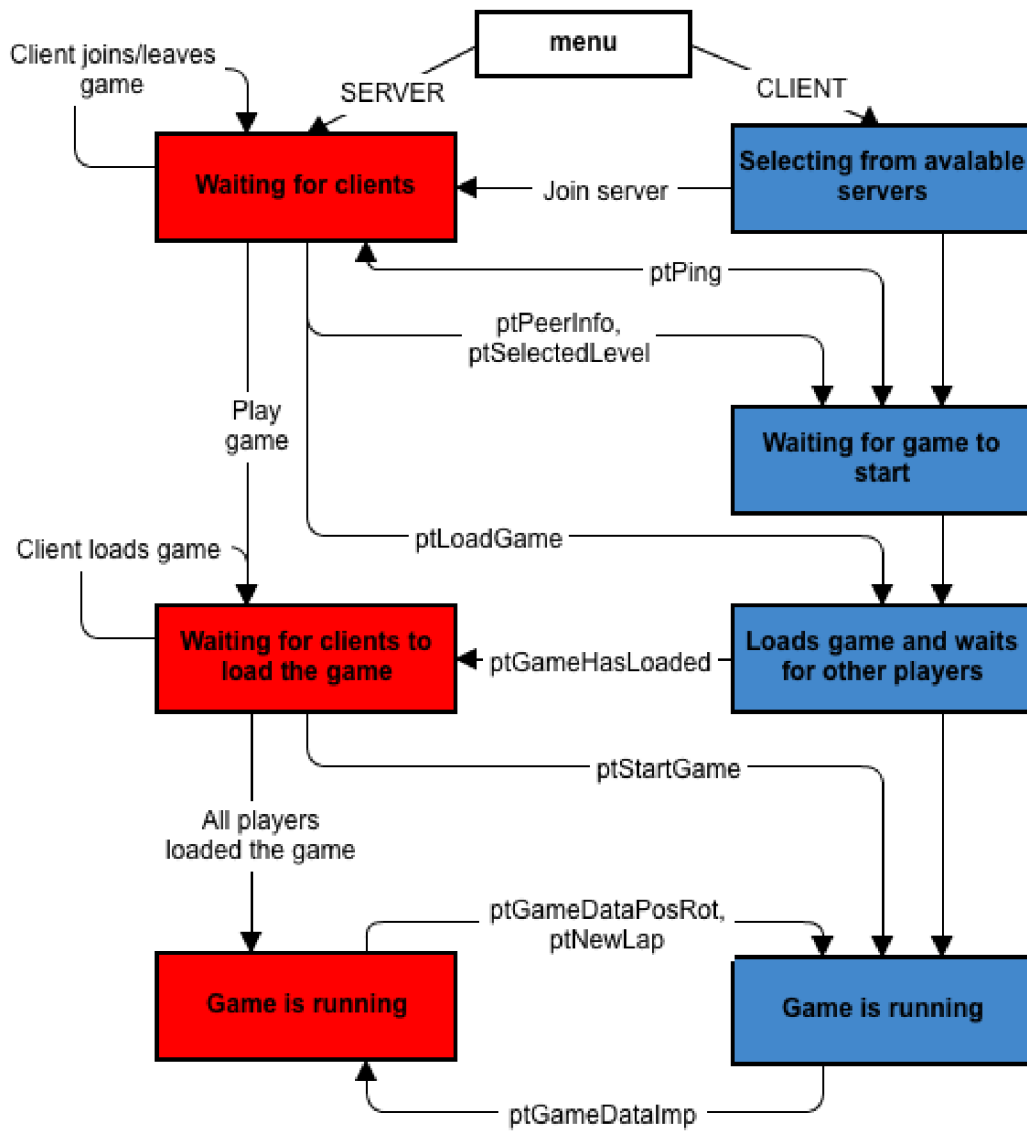
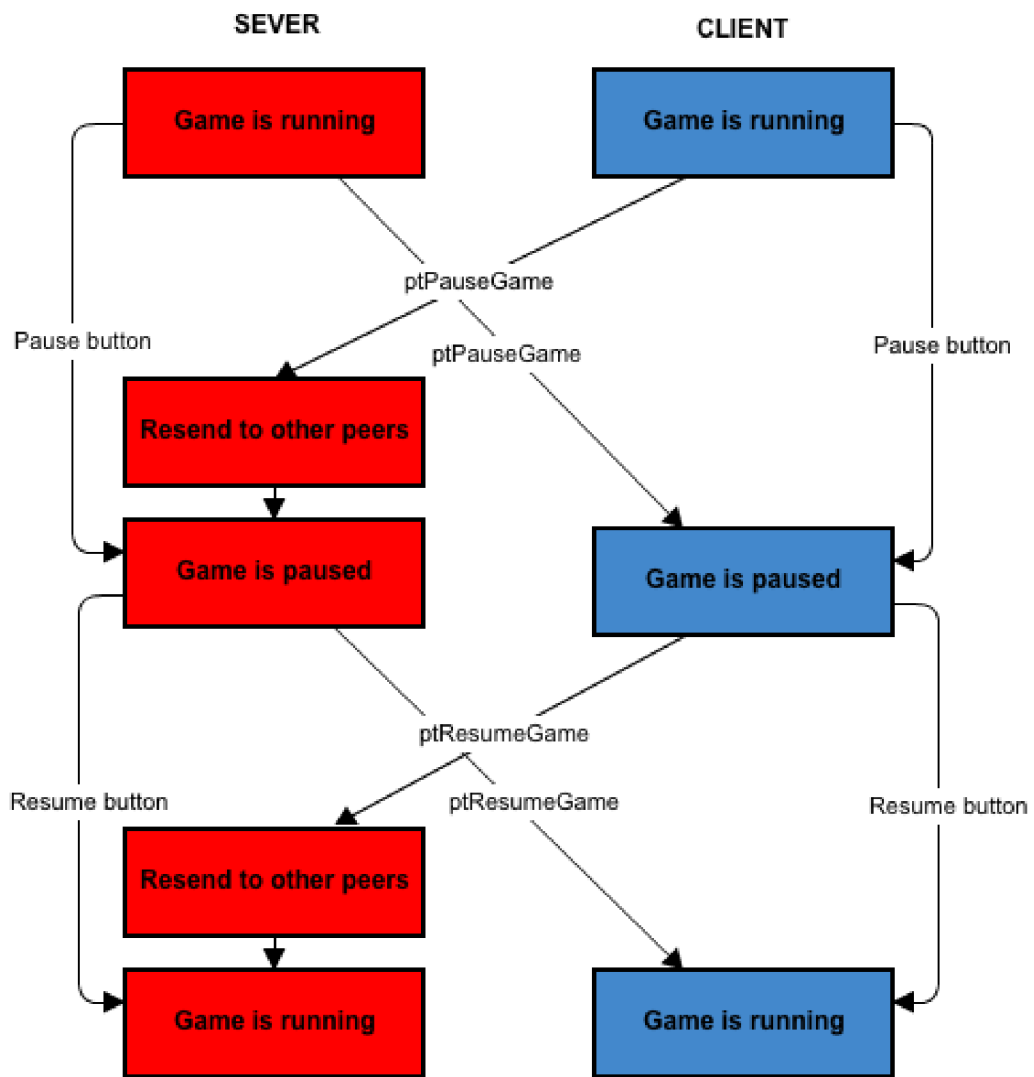Figure 4.13: Client-server communication

Figure 4.14: Pausing and resuming the game

4.5   Level creator

Level creator is a multitouch tool built into the game. Players can use it to create new levels. Each level is basically a grid of tiles. This grid has constant size of 5 rows and 7 columns.

4.5.1   Interface

Interface of the level creator consists of three screens. The first screen 4.15 contains a list of already created levels. User can choose to open or to edit any of the levels from the list, or he can create a new level. Screen 4.16 serves for creating a new level or editing information about an existing level. User can edit name of the level and number of laps. The final and most important screen of the level creator serves for the actual level construction. 4.17 Its interface has three parts:

- **Level layer** shows how the level would look like. User can zoom in/out and scroll through this layer in both directions.

- **Available tiles scroller** contains all the tiles, that can be used to compose the level.

- **Menu** offers navigation options, and options to save the level or to cancel changes.
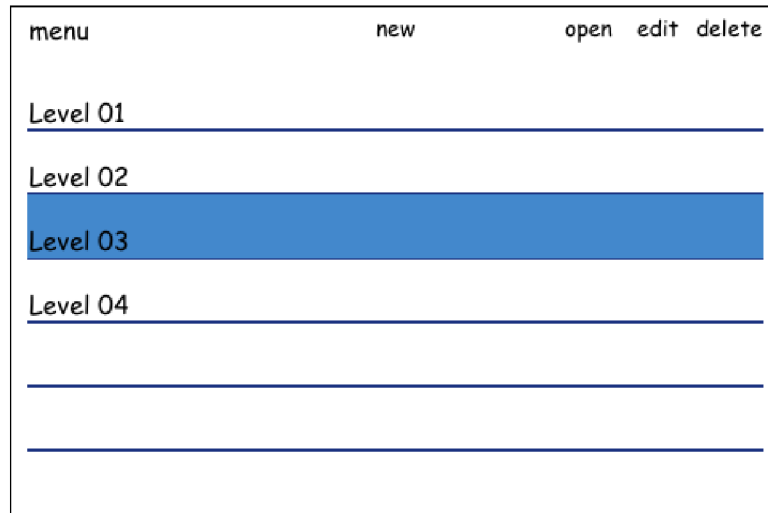

Figure 4.15: User interface of the level creator

Figure 4.16: User interface of the level creator

Figure 4.17: User interface of the level creator

4.5.2   Defining a new level

User composes a new level in the main creator screen 4.17 described in 4.5.1. User can compose tiles however he likes, although one condition has to be met. Level has to contain the tile with the start line 4.18, 4.19, or else it is marked unfinished and cannot be played. The work with tiles is described below.
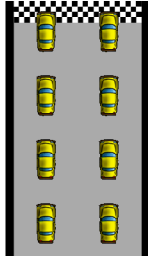

Figure 4.18: Vertical start line tile   Figure 4.19: Horizontal start line tile

1. User taps on a tile and holds finger for a while

2. When the tile becomes active he is able to move it around

3. Possible following actions depend on the type of the tile

   - Tile from the scroller was released above:
     - the scroller, nothing happens
     - a free map cell, it is placed to this cell
     - an occupied map cell, it is placed to this cell and the occupying tile is removed
   - Tile from the tile map was released above:
     - the scroller, it is removed from the map.
     - a free map cell, it is moved to this cell If is this cell occupied by other tile, this is removed from the map.
     - an occupied map cell, the two tiles are swapped

4.5.3   File format

Each level is saved to a individual property list file. Property list files are files for storing serialized objects in key-value form. Levels created by user are stored separately to a documents directory of the application. Predefined levels are stored in applications main bundle, this prevents user from manipulating with them. Level property files contain:

- Two-dimensional array of tile types

- Initial position of each car

- Initial rotation of the cars

- Best lap time

- Best track time

- Position of the start/end line

Chapter 5

Implementation

The purpose of this chapter is to provide the reader with description of the interesting parts of the implementation and to summarize user tests and survey. It describes the structure of the project and the application. In this chapter are also listed and described all the imported classes and the tools used in development.

5.1 Application structure

Besides framework sources game consists of 44 classes, 40 are created in thesis, two are predefined by Xcode's application template and two are imported. To keep so many classes organized it is inevitable to define a logical structure. Even though Objective-C doesn't support package systems similar for example to Java, it enables to create source groups structure 5.1. Name of every created class has prefix **BC** to help distinguish them from other sources.
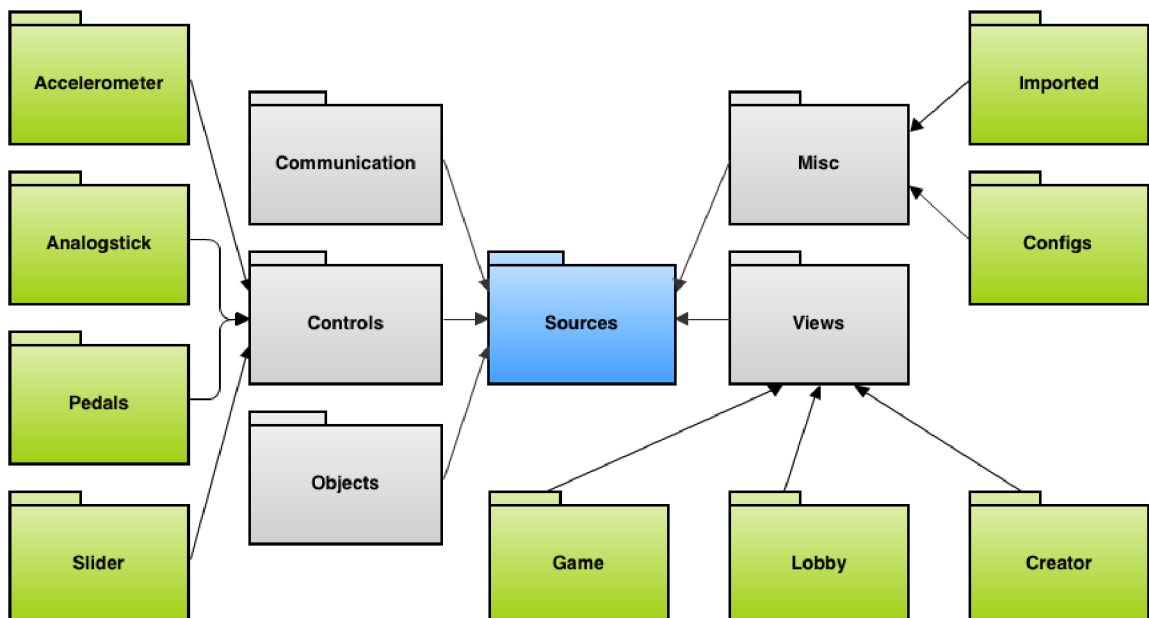

Figure 5.1: Source groups hierarchy

5.2   Game manager

As its name suggests this object manages the flow of the game. It is a shared singleton object accessible from anywhere in the game. Its main task is implementing scene-swapping logic. Transitions between scenes are displayed in the graph 5.2 It also encapsulates and provides access network communication adapter and informations about application and game states.
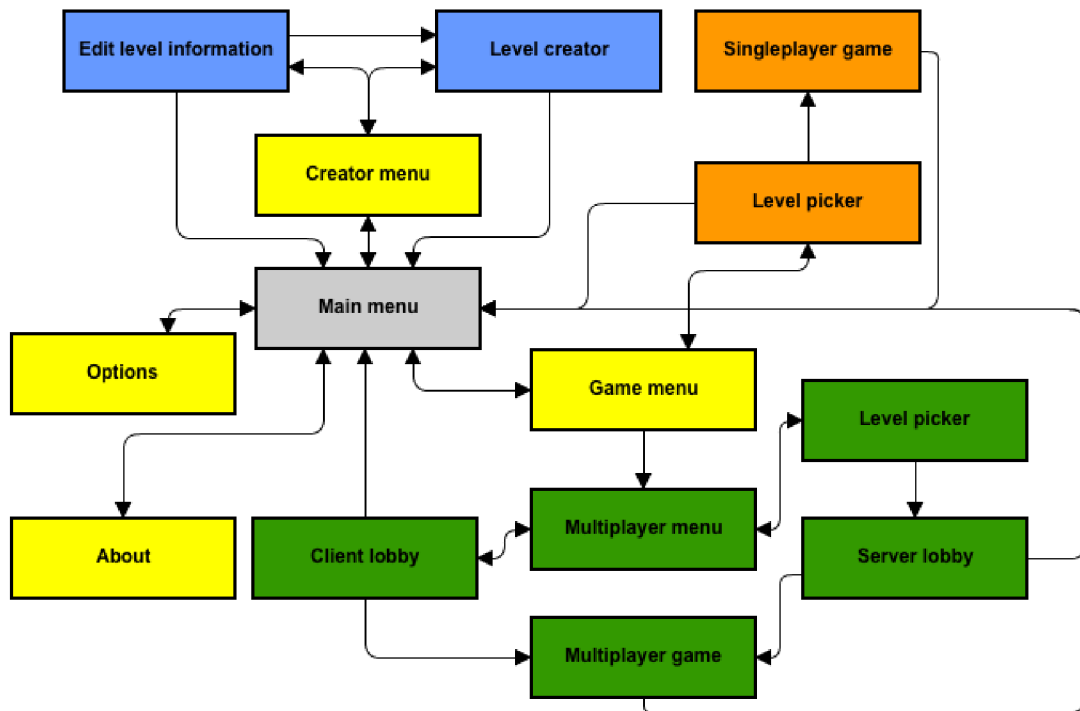

Figure 5.2: Scene transitions graph

5.3   Packets

Content of individual game packet types is very different. Some packets contain just simple values, e.g message packets contain only one integer value defining the type of the message. On the other hand some packets contain complex Objective-C objects, e.g *ptPeerInfo* packets contain encoded *NSArray* of integer values representing car positions on the start line. As was mentioned on the page 6 packets has to be *NSData* objects. Apple provides simple conversion between *NSData* and *NSString* objects. All packet's data are therefore firstly encoded to string and then to *NSData*. Individual pieces of data are separated by separator sign: "# ".

5.4   User preferences

In options screen user sets his game preferences. He can choose what controls he wants to use and whether the music should play or not. Apple provides a API called *NSUserDefaults* for

storing simple setting values like these. This API stores all settings in key-value form inside the application bundle. Controls type is save in form "selected_controls"=>"controls_type"and music preferences are saved as "music"=>"on/off".

5.5 Imported sources and resources

UIKit objects wrapper

Cocos2d miss some basic user interface elements, which sometimes makes using *UIKit* interface elements inevitable. However they cannot be used in the *CCNode* hierarchy directly. They have to be contained inside a *CCNode* wrapper. Implementation of such a wrapper is described in the *Cocos2d for iPhone 1 Game Development Cookbook* [25]. Similar implementation is also available as a cocos2d extension. I use this extension in level creator, level picker and lobby scenes to embed *UITableView* and *UITextField*.

PhysicsEngine Box2D loader

Is a class that helps to initialize Box2D fixtures created in PhysicsEngine. It is shipped with the tool and free to use and modify.

Car sprites

Car sprites used in the game were created as study materials to be used in a game development workshop. They can be found on CSTA[1] website. [1] And are shared without any license, completely free to use.

Music

Music playing in the game is a creation of Matthew McFarland. Together with his other works it can be found at his website. [21] His music is shared under creative commons license.

5.6 Tools

Gimp

Most of the graphics content in the game is original, created in Gimp. These graphics include: application icon, splash screen, main menu, some of the buttons, level tiles, borders of the track and finish line.

Zwoptex

Zwoptext was used for packing the textures created in Gimp.

PhysicsEditor

Every fixture in the game was created in PhysicsEditor. I used it due to he fact, that almost every game fixture consist of multiple polygon shapes.

---

[1]The Computer Science Teachers Association of Oregon

34

Xcode and Instruments

The game was developed in Xcode without support a of ARC [2]. Tracking reference count via `retain` and `release` messages tends to cause errors and unexpected behavior. To help me resolve some of these problems I used debugging tool in Xcode and Instruments.

Gliffy and Pencil

Gliffy is a online diagram creation tool. Pencil is a free multi-platform tool for designing user interface. They were used them to design parts of the application and to create graphic materials for this technical report and the presentational poster.

5.7   Requirements and testing

Development was done in Xcode 4.1, on a second generation iPod Touch and on a iPhone 4. There may be compilation problems in other versions of XCode due to third-party frameworks. Game requires iOS version 4.1 or higher. I implemented a system measuring the packet exchange latency. Latency of a packet depends on the count of packets sent per unite of time. When the game is not running and game information packets are not being sent, latency of a packet is under 100ms 5.3. However when the game is running, information packets are being sent 60 times per second. This additional load causes huge increase in the latency. Latency also depends on which device is server and which is client. In-game latencies are show on chart 5.4 and 5.4.
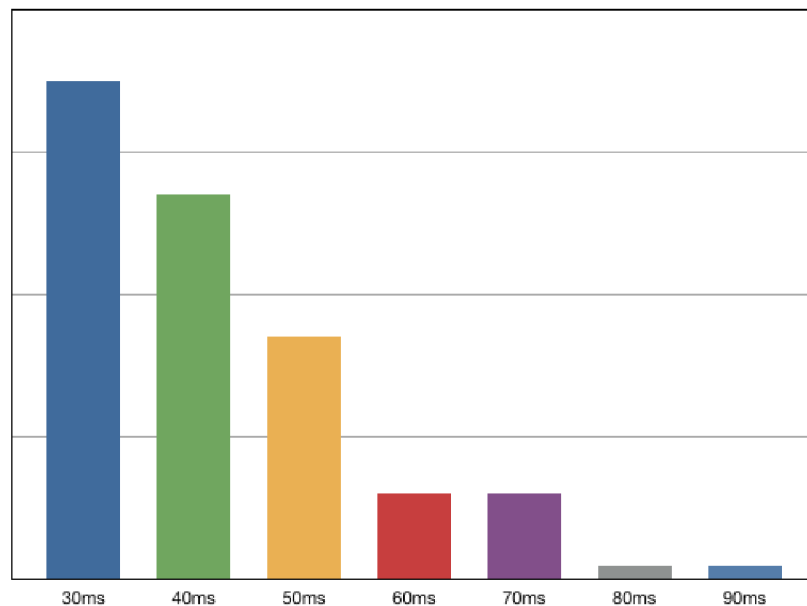

Figure 5.3: Packet latency distribution in the game lobby

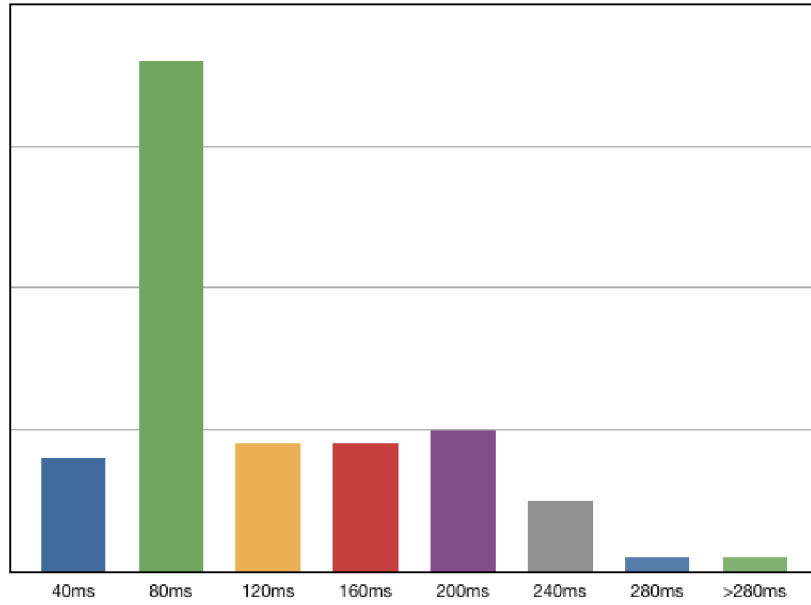[2]Automatic Reference Counting system, available since iOS 5

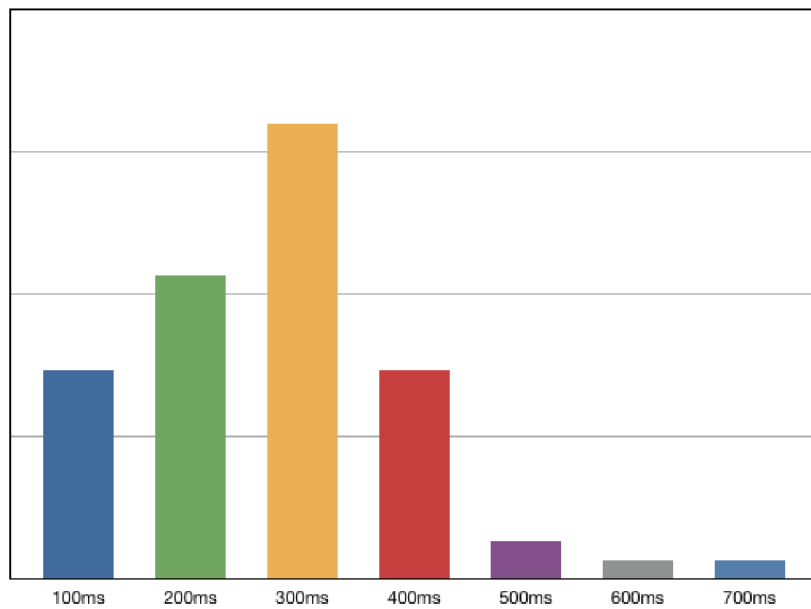Figure 5.4: Packet latency distribution, iPod 2nd gen. server, iPhone 4 client

Figure 5.5: Packet latency distribution, iPhone 4 server, iPod 2nd gen. client

5.8 User tests and survey

The game has been tested by multiple people during the whole development process. Based on their suggestions, car handling was adjusted, several alert screens added and work with *UITables* altered. Furthermore a final user test and survey was organized. The test attended five people. Under supervision they played the single-player as well as the multiplayer, they tried all controls, and worked with the level creator. After the test they answered few survey questions and suggested ideas to improve the gaming experience. Figures 5.6 and 5.7 display results of the survey.
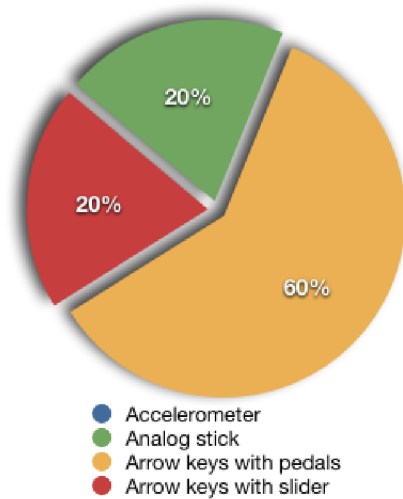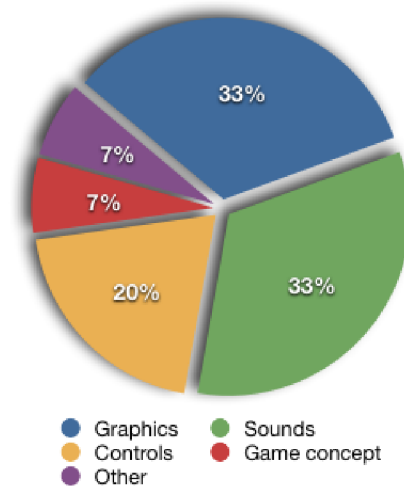

Figure 5.6: Player's preffered controls

Figure 5.7: Suggested improvements

Chapter 6

Conclusion

This is the last chapter of the thesis. It summarizes the whole work and development process. It reviews achieved results and discuss possible improvements and future development.

6.1 The work progress

Work on the thesis progressed according to its specification. Since I had no previous experience with the iOS platform I began with the study of its APIs [1] and Objective-C programming language. After I learned the basics I moved to study the third-party frameworks. I focused on the open-source frameworks which could be used to create the game. Results of these studies were concluded as a part of the term project and are expanded and completed in the first two chapters of this theses. When I knew the capabilities of the frameworks I began working on the concept and design of the game. It was already mentioned throughout the thesis, that the game controls are an essential part of the gaming experience. Therefore the main goal was to create an entertaining game that will provide multiple different controls for players to choose from. Moreover this allowed to create a survey studying preferences of the players. This study is described in the previous chapter. Most important parts of the game are described in the chapter design on page 17. Application structure and implementation details are mentioned in the chapter implementation on the page 32. The last part of the specification of the thesis implies uploading the game to the *AppStore* [2]. Only programmers registered under the paid *iOS Developer Program* are able to upload a game. Since this game was developed under the *iOS Developer University Program*, it cannot be uploaded. Therefore I created a website [24], where anyone can download the game for free.

6.2 Resulting application

The game created in this thesis is meant to provide competitive entertainment, for small groups of players, similar to games like *Worms* or *Bulánci*. The user interface of the game together with the controls was designed to be intuitive and responsive. Application reacts to every user action, whether it is correct, or not. Another significant part is the multiplayer for up to eight players. The bluetooth network proved to be insufficient to provide communication for an action game. An action game requires a high frequency

---

[1] Application programming interface
[2] An online store containing applications available for iOS devices

of the packet exchange. Connected devices are not able to process incoming and outgoing messages quickly enough. As a result the network latency is too high for the game to be fluid. Although the game is still playable, it aggravates the experience. The user survey proved the level creator tool to be intuitive and fun to play with. It is due to its transparent interfaces and support of the touch gestures known from other iOS applications. In conclusion, the created game differs from the other games for iOS by implementing various game controls, available local multiplayer and the level creator. However to fulfill its potential it needs some improvements. They are discussed in the next section.

6.3   Possible improvements and future development

The future development of the game should focus on improving the network communication. The bluetooth communication should be replaced with the local wireless network. However since not all players have companions to play with, game could include the *Game Center* support and the online multiplayer. According to user survey graphics should be also improved and more sound effects added. Another improvement could be option to calibrate the game controls, so they suit the individual style of each player.

Bibliography

[1] Class materials. http://webclass.superquest.net/gamemaker-projects/PROGRAM/Sprites/Highway_pr/. [online].

[2] Alasdair Allan. *Learning iOS Programming*, chapter 3. Your First iOS App. O'Reilly, second edition, 2012. [cit. 2012-04-22].

[3] Alasdair Allan. *Learning iOS Programming*, chapter 9. Using Sensors. O'Reilly, second edition, 2012. [cit. 2012-04-22].

[4] Löw Andreas. Physicseditor. http://www.physicseditor.de/. [cit. 2012-04-22], [online].

[5] Apple Inc. Game kit programming guide. https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/GameKit_Guide/Introduction/Introduction.html. [cit. 2012-04-22], [online].

[6] Apple Inc. Game Kit Programming Guide game center overview. https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/GameKit_Guide/GameCenterOverview/GameCenterOverview.html#//apple_ref/doc/uid/TP40008304-CH5-SW7. [cit. 2012-04-22], [online].

[7] Apple Inc. Game Kit Programming Guide, peer-to-peer connectivity. https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/GameKit_Guide/GameKitConcepts/GameKitConcepts.html#//apple_ref/doc/uid/TP40008304-CH100-SW1. [cit. 2012-04-22], [online].

[8] Apple Inc. Game Kit Programming Guide, sessions. https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/GameKit_Guide/Art/GKSession_communication.jpg. [cit. 2012-04-22], [online].

[9] Apple Inc. iOS App Programming Guide. [cit. 2012-04-22], [online].

[10] Apple Inc. ios human interface guidelines. https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html. [cit. 2012-04-22], [online].

[11] Apple Inc. ios technology overview.
https://developer.apple.com/library/ios/#documentation/Miscellaneous/
Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html. [cit.
2012-04-22], [online].

[12] Apple Inc. The objective-c programming language.
https://developer.apple.com/library/ios/#documentation/Cocoa/
Conceptual/ObjectiveC/Introduction/introObjectiveC.html. [cit. 2012-04-22],
[online].

[13] Apple Inc. Opengl es programming guide for ios.
https://developer.apple.com/library/ios/#documentation/3DDrawing/
Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html.
[cit. 2012-04-22], [online].

[14] Apple Inc. Uiacceleration class reference.
https://developer.apple.com/library/ios/documentation/UIKit/Reference/
UIAcceleration_Class/Art/device_axes.jpg. [cit. 2012-04-22], [online].

[15] Apple Inc. Uiacceleration class reference.
https://developer.apple.com/library/ios/#documentation/UIKit/Reference/
UIAccelerometer_Class/Reference/UIAccelerometer.html. [cit. 2012-04-22],
[online].

[16] Catto Erin. Box2d user manual. http://box2d.org/manual.pdf. [cit. 2012-04-22],
[online].

[17] Gamua. The sparrow manual.
http://wiki.sparrow-framework.org/manual/start. [cit. 2012-04-22], [online].

[18] Gamua. The sparrow manual. http://wiki.sparrow-framework.org/_media/
manual/class_hierarchy_v1_1.png?w=760. [cit. 2012-04-22], [online].

[19] Gamua. The sparrow manual.
http://wiki.sparrow-framework.org/_media/manual/coordinate_systems.png.
[cit. 2012-04-22], [online].

[20] Howling Moon Software. Chipmunk game dynamics.
http://chipmunk-physics.net/release/ChipmunkLatest-Docs/. [cit. 2012-04-22],
[online].

[21] Matthew McFarland. Mcfarland beats.
http://www.mattmcfarland.com/index.php?option=com_content&view=
article&id=195:stunt&catid=36:exciting&Itemid=55. [online].

[22] Daley Michael. *Learning iOS Game Programming*, chapter 15. Collision Detection.
Addison-Wesley Professional, 2010. [cit. 2012-04-22].

[23] Ray Wenderlich Michael Daley, Rod Strougo. *The IOS Game Programming
Collection*, chapter 9. Tile Maps. Addison-Wesley Professional, 2012. [cit.
2012-04-22].

[24] Tomáš Mičiak. Drift races. http://ios-races.webs.com/. [online].

[25] Burba Nathan. *Cocos2d for iPhone 1 Game Development Cookbook*, chapter 5. Scenes and Menus. Packt Publishing, 2011. [cit. 2012-04-22].

[26] Quesada Ricardo. cocos2d for iphone wiki. http://www.cocos2d-iphone.org/wiki/doku.php/. [cit. 2012-04-22], [online].

[27] Quesada Ricardo. cocos2d for iPhone wiki programming guide. http://www.cocos2d-iphone.org/wiki/lib/exe/fetch.php/prog_guide:layers.png. [cit. 2012-04-22], [online].

[28] The Khronos Group. Opengl es. http://www.khronos.org/opengles/, 2012. [cit. 2012-04-22], [online].

[29] Lindeijer Thorbjorn. Tiled map editor. http://www.mapeditor.org/. [cit. 2012-04-22], [online].

[30] Lidholt Viktor. Cocos builder. http://cocosbuilder.com/. [cit. 2012-04-22], [online].

[31] ZWOPPLE Creative Software. Zwoptex. http://zwopple.com/zwoptex/. [cit. 2012-04-22], [online].