Death to Balloons

Semester Thesis

Alexander Waldin
waldina@ee.ethz.ch

Advisors:
Johannes Schneider, Samuel Welten

Supervisor
Prof. Dr. Roger Wattenhofer

Distributed Computing Group
Computer Engineering and Networks Laboratory (TIK)
Department of Information Technology and Electrical Engineering

June 2011

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Abstract

Background: In recent years the Android mobile phone platform has continued to gain market share. Google's operating system supports a variety of different sensors to allow the phone to interact with the user and the environment. Despite this freedom, many current video games seldom attempt to incorporate these sensors and prefer to rely on tried and true input mechanisms. In this project we experiment with the sensors and come up with innovative ways of interacting with objects in a game.

In the game "Death to Balloons", we implemented several different control schemes using a variety of sensors such as touch, acceleration, audio, etc. to control a balloon on the screen. We also use Bluetooth to permit two players with smartphones to compete against each other using these input schemes.

Table of Contents

1. Motivation

1.1. Introduction

Video games on cell phones have come a very long way since the beginning of the 21st century. In the early days cell phone games were limited by the hardware of the older mobile phone models and were often very simple black and white games such as Snake. These games came pre-installed on the cell phone and adding more games was either very difficult or impossible. As mobile phones became ever more powerful, so did video games. Today, mobile video games can take up quite a bit of memory and may sport graphics comparable to a 5th generation game console such as the PlayStation. The mobile games industry is large, topping $800 Million in revenue in 2010 in the United States alone (1). Two recent developments in the past few years have revolutionized the mobile game industry, the introduction of smart phones and the availability of online software stores, for example, the Android Market developed by Google and the App Store maintained by Apple.

Smartphones are mobile phones that offer greater connectivity capabilities and computing power than a contemporary feature phone. Furthermore, they have a touchscreen and often a variety of sensors such as acceleration sensors to detect movement, magnetic sensors to determine special orientation, light sensors to adjust to various lightning situations and to act as a primitive proximity sensor. Most smartphones also have a GPS to pinpoint its location.

With the capabilities of smartphones listed above, it should be possible for modern mobile games to feature brilliant graphics and intuitive controls. Developers can cleverly use the large number of sensors smartphones put at their disposal instead of cumbersome control schemes using the touchscreen, or worse, the tiny keyboard. It is certainly the case that some game developers have attempted to incorporate alternative control schemes into their games, for example, in a game called Space Willi the player controls a space ship with voice commands (2). However despite this effort innovative controls are rare. Many of the top selling games use a very basic control scheme using just the touchscreen and buttons (3), as do all of IGN's top ten games of 2010 (4). Angry Birds, which has been called "the largest mobile app success the world has seen so far" uses very simple touch controls (5). Clearly, there is still much experimentation required to develop good controls that use alternative input methods. The goal of this semester project was to develop a game that uses the variety of sensors that a smartphone, such as the Nexus 1, puts at a developers disposal.

1.2. Platform

Android is an open source, Linux-based smartphone OS. We chose the Android Platform for several reasons:

1. *Open Source:* Because Android is an open source system it is possible to access the code that drives the system and to understand the platform's API.
2. *Programming Language:* Unlike the iPhone, which uses Objective-C, a language that is not used for anything beyond development for Apple's products, Android uses Java, a professional language that has been around since the 90's and for which there are many resources and excellent IDE's such as Eclipse.

1.2.1. An Overview of the Android Platform

Android is built on top of a solid foundation, the Linux Kernel, which provides a hardware abstraction layer for Android and allows Android to be ported to a variety of platforms. Android itself uses Linux for memory management, networking and other operating system services. However, programs written for Android never make Linux calls directly.

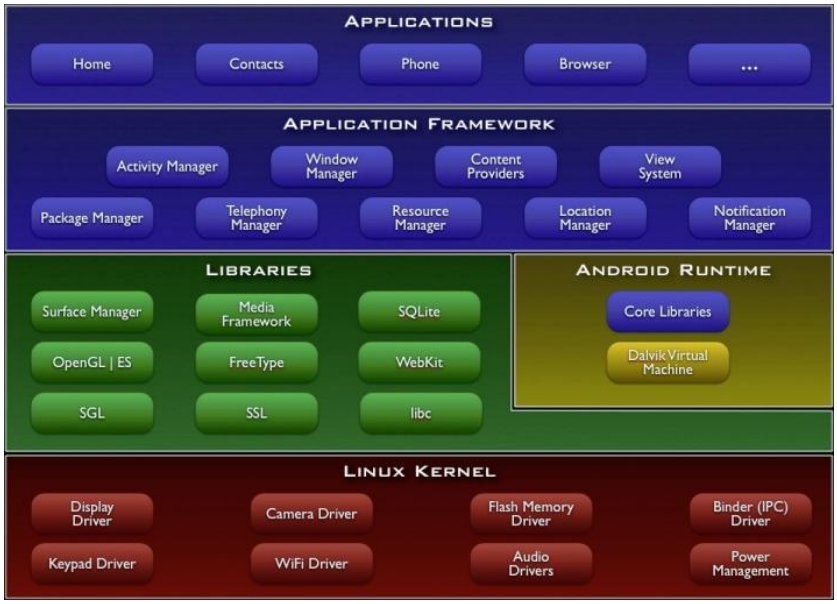

Figure 1-1 Android Architecture (8)

On top of the kernel lies the Android runtime, which contains the Dalvik virtual machine, a Java virtual machine optimized for low memory requirements. Above this is the Application Framework layer, which provides high-level building blocks for use by applications, for example, the Activity Manager, which controls the life cycle of applications. The highest layer is the Applications layer. This is the layer in which most programs are written and is also the layer in which our game was written.

1.3. Requirements and Design Criteria

The following lists the requirements and design criteria that were decided upon at the beginning of the project:

Functionality

The game should be a multiplayer game and support at least two people. Multiplayer should be local and implemented either through Bluetooth or Wireless Local Area Network. The target device for the game should be the HTC Nexus one, and the game should take advantage of the many sensors this smartphone provides for implementing a variety of innovative control schemes.

Graphics and Gameplay

The game's focus is on implementing control schemes that take advantage of the smartphones sensors. Graphical prowess is not a goal for game design. For the same reasons, the gameplay, i.e., the specific way in which players interact with the game, should be kept simple. Furthermore, simple gameplay makes it is easier to experiment with control schemes because they can quickly be implemented and play-tested.

Other

User Interface: The User Interface should be kept simple and intuitive.

Speed: Because the game is a multiplayer game, which takes place in real-time, the reaction time of the game should be fast. A player should not experience any lag due to the input controls or the wireless communication between the devices.

Software E*xtensibility*: The focus here lies on two things. First, it should be possible with little effort to extend the game with new control schemes and second, it should be possible to add more players later.

1.4. Realization: Introducing Death to Balloons

Death to Balloons is a multiplayer-only game for two players. The player that initiates the game is referred as the host. The host invites a challenger to play. Communication between the host's and challenger's smartphones is via Bluetooth. Players take turns playing the roles of adversary and survivor. There is one balloon, and the adversary tries to pop the balloon by driving it into the spikes at the edge of the screen. At the same time, the survivor tries to protect the balloon from being popped by steering it away from the spikes. In each round each player takes on the role of adversary and survivor exactly once. Each round consists of the following stages:

1. First the host plays as the adversary and the challenger plays as the survivor. Only the survivor can increase his score. He does this by keeping the balloon alive for as long as possible, that is, as long as the balloon is alive, his score increases. Thus, it is in the interest of the adversary to pop the balloon.
2. Once the host succeeds in popping the challenger's balloon the roles switch. Now the host plays as the survivor, trying to raise his score by keeping the balloon alive for as long as possible. Once the adversary pops the balloon a new round begins and once again the host is the adversary and the challenger the survivor.

This process repeats itself for a set of rounds determined by the host at the beginning of the game. Once the game is over, the scores from each round are tallied up and the player with the highest score, and thus kept his balloon alive for the longest amount of time, wins.

Because it is often difficult for a player to keep track of whether or not he is currently playing the role survivor or adversary, the balloon is color coded to reflect the player's role. The adversary's balloon is red, and the survivor's is blue. Thus, a player can keep the following simple rule in mind: *Blue balloons are good, keep them alive; red balloons are bad, destroy them.* Figure 1-2 is a screenshot of what the survivor sees when a new round begins. His goal is to prevent the balloon from being driven into the spikes, which are all along the edge of the phone. For him the balloon appears blue, indicating he should protect it.

Figure 1-2 A screenshot of Death to Balloons (9)

1.4.1. Control Schemes

The game offers a variety of control schemes to choose from. The host picks the control scheme for both the adversary and the survivor when he initially sets up the game. Although it is possible to choose the same control scheme for both roles, it is encouraged to choose different ones, as this leads to more game variety.

The control options that are at a player's disposal are:

1. *Touch Control* – This is a simple control scheme. The player controls the balloon by touching the screen. The balloon will accelerate in the direction of his finger. The further away he places his finger from the balloon, the faster the balloon will accelerate.
2. *Air Bubble Control* – A simple control scheme that uses the orientation sensor to move the balloon. The player tilts the phone and the balloon moves like the air bubble in a spirit level. The steeper the player tilts the phone, the faster the balloon will accelerate.

3. *Jolt Control* – A control scheme which uses the acceleration sensor in an attempt to detect in which direction the player moves his phone. By quickly moving (jolting) the phone forward, backward, left or right, the player can move the balloon in the direction he jolts the phone. The quicker he jolts the phone, the greater the force that acts on the balloon.
4. *Wind Compass Control* – A control scheme that uses the orientation sensor as well as the microphone to simulate wind acting on the balloon. A player makes a "shhh" sound to generate the wind; the louder he makes this sound, the stronger the wind will be. To influence the direction in which the wind blows, the player rotates the phone around the z-axis[1]. The wind will always blow away from the player. To aid the player, a cloud indicates the direction in which the wind blows. Figure 1-3 shows the *Wind Compass* in action. The cloud indicates that the balloon will go down when the player makes a "shhh" sound. The balloon is red, which means the player is in the role of the adversary whose goal is to pop the balloon. In addition, the balloon's expression has changed from happy to frightened, signifying that the balloon is close to the edge.
5. *Wind Roll Control* – A control scheme that is similar to the Wind Compass control. Instead of rotating the phone around the z-axis, the player rotates the phone around the y-axis. This control scheme allows the player to control the direction in which the wind blows more precisely, however, it is less intuitive.
6. *Bomb Control* – The final control scheme option uses a combination of the touch screen, the acceleration sensor and the light sensor. The player places up to three bombs with his finger. Once a bomb is placed it will take five seconds to arm itself. Before the bomb has been armed it is black, once it arms itself, it will change its color depending on whether or not the player or his opponent placed the bomb. If the player placed the bomb, it will turn blue, if the opponent placed the bomb it will turn red. A player can set off all bombs he has placed and that have armed themselves by shaking the phone. When bombs explode they push the balloon away from them; the force depends on the distance from the bomb to the balloon. A balloon directly on top of a bomb will receive a large force, and the force drops quickly the further the balloon is from the bomb. Furthermore, once each round, a survivor can gain some breathing space by covering the light sensor with his hand. This results in all the bombs on the screen being erased. Figure 1-4 illustrates the bomb controls: Three balloons have been placed, the black one is still inert and two have armed themselves. The fact that these two bombs are red indicates that they were placed by the opponent.

[1] See Figure 3-1 for the orientation of the smartphone's axes.

Figure 1-3 A screenshot depicting the *Wind Compass* control scheme

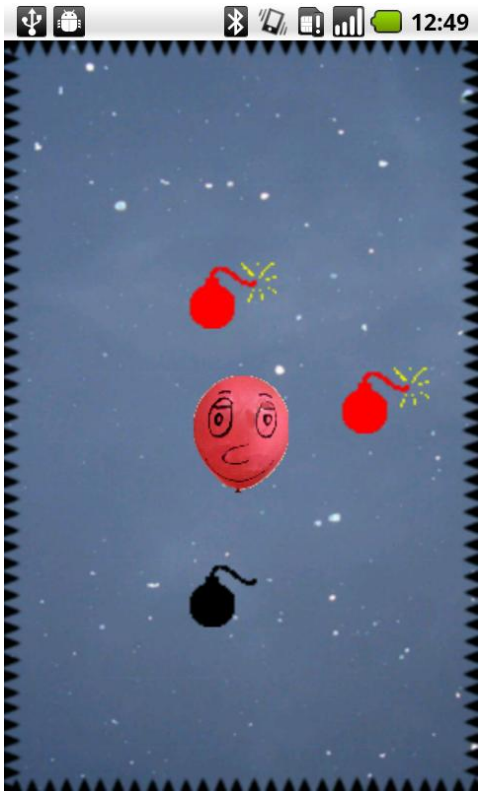

Figure 1-4 A screenshot showing the *Bomb* control scheme in action

2. Software Architecture

2.1. Activities

Death to Balloons has four different activities; where an activity is a type of Android component that provides a screen with which users can interact. They are:

1. *DeathToBalloons* – This activity displays the main menu.
2. *SettingsScreen* – This activity displays the screen where a user can choose the control schemes for both the adversary and the survivor, and where he can set the number of rounds.
3. *Connection* – This activity displays the screen while the connection is being set up. It lists all discovered phones.
4. *AbstractGame* – The classes that inherit from this class implement activities that display the game-screen.

2.2. System Overview

The classes of Death to Balloons are roughly divided into three categories (see Figure 2-1):

Classes dealing with the Initial Setup

The classes *DeathToBalloons* and *SettingsScreen* handle the initial setup. All setup parameters, for example, the control schemes, the number of rounds and whether or not the game should be set up as a Bluetooth master or slave, are specified here. These parameters are stored statically in the *ResourceLookup* class.

Classes handling the Connection

Once the user has set the initial parameters it is up to the *Connection* class to connect to the other phone. If a user is hosting a game, *Connection* will ask the user to enable Bluetooth and make the phone discoverable. It then opens a server port and listens for phones attempting to connect. If a user is attempting to connect as a slave, the phone will search for Bluetooth devices in the area. All devices it discovers will be listed for the user. The user must then select a corresponding partner. Once a connection has been established, the *Connection* activity then spawns a thread that supports an object stream. This is an instance of the *ConnectedThread* class, and a reference to it is stored in the *ResourceLookup* class.

Classes handling the Game

Depending on whether the game is set up as a host or challenger, the activity *GameHost* or *GameSlave* will be started. Both activities set up and tear down the actual game. Furthermore, they initialize and register all listeners for the sensors. Because all processing such as updating the game's state and interpreting the sensor's inputs is handled by the host, the class *GameHost* is more complex.

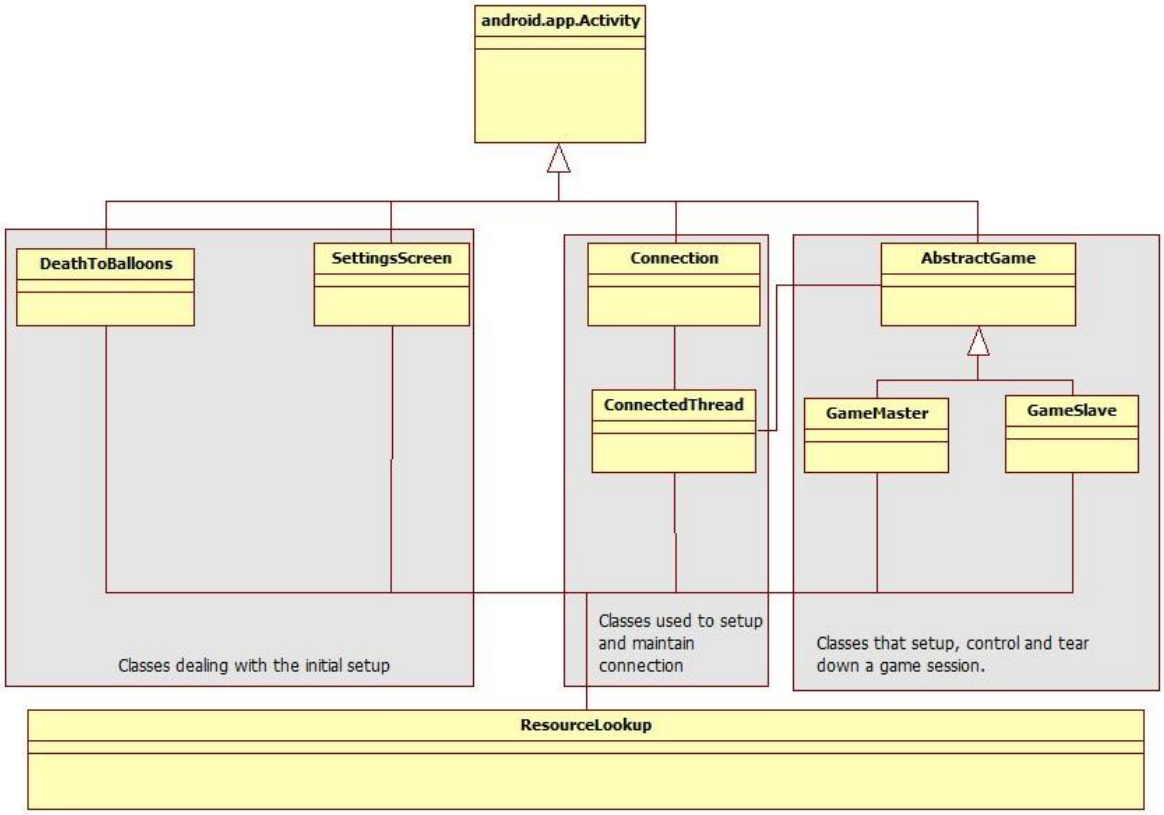

Figure 2-1 A System Overview

2.3. The Game Framework

The major classes of the game framework, which are set up by either the *GameMaster* or *GameSlave* activity at runtime, can be roughly described by the Model-View-Controller architecture. The corresponding classes/interfaces are: *Model*, *GameView* and *IController*. The classes that implement the IController interface are quite different and depend on whether the game is set up as a master or slave.

2.3.1. The Game Framework in Host Configuration

When a player's smartphone is set up as a host, there will be instances of the following classes: *Model*, *GameView*, *Controller* and *StandInView*.

Model

The *Model* holds all information about the current state of the game, such that for the balloon, the players and the current round. If the player is using a control scheme that displays a cloud or bombs, the position and state of these will also be stored in the model. Furthermore, because the game is running in host configuration, the model runs a thread that continuously updates the current state of the game and applies all forces to the balloons.

GameView

The *GameView* class is responsible for displaying the graphics to a user. A thread running in the GameView class polls the Model class instance every 15 ms for the current state of the game. The GameView is responsible for enabling and disabling the static sensor classes depending on which

control scheme the player is currently using. Furthermore, because the game is in host configuration, *GameView* is responsible for detecting when the balloon is driven into the spikes.

Controller

In the host configuration, there is a single instance of *Controller* that takes input from sensors and uses this input to update the model. Each sensor has a reference to this controller instance and sends its input to the controller by calling the method *updatePhysics(ISensorInput input)*, which is defined in the interface *IController*. The controller maintains a set of helper classes that can interpret the input from a given type of sensor. The controller updates the model by examining the type of input it receives and then forwarding this input to the corresponding helper class. For details on how the Sensor-Input-Controller system works see Section 2.4.

StandInView

The *StandInView* class is responsible for keeping the challenger up to date on the state of the model. It does this by polling the *Model* class instance and sending the model information to the challenger using methods defined in the *ConnectionThread* class.

2.3.2. The Game Framework in Challenger Configuration

When a player's smartphone is set up as a challenger, there will be instances of the following classes: *Model*, *GameView* and *StandInController*.

Model

As in the host configuration the model keeps track of the current state of the game. However no model thread is being run. Instead the instance of *ConnectionThread* continuously updates the model with the information it receives from the host.

GameView

The instance of the GameView is almost exactly the same as in the host configuration. However, it does not keep track of balloon-collisions.

StandInController

As in host configuration all sensors require an instance of the *IController* interface to which they report sensor information. In the case of a challenger, this is an instance of *StandInController*. *StandInController* forwards all sensor information reported via calls to *updatePhysics(ISensorInput input)* to an instance of *ConnectedThread*, which in turn sends them to the host.

2.3.3. Communication Scheme

A scheme depicting how the instances of the classes interact with each other across both platforms can be seen in Figure 2-2.

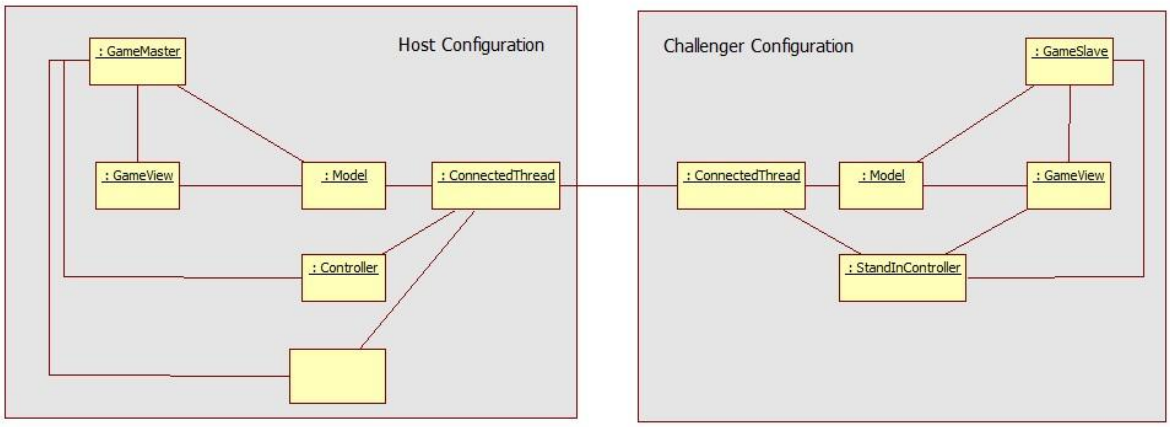

Figure 2-2 Game Framework at Runtime on both Phones

2.4. The Sensor–Input–Controller System

A key requirement for the implementation is that it can be easily extended with new control schemes. This is achieved by using the strategy design pattern. In order to add a new control scheme one simply writes three classes: A sensor class, a sensor input class and a controller helper class.

A sensor turns a user's actions into an instance of an input class that implements the *ISensorInput* interface. The sensor input is then forwarded to an implementation of the IController interface, which in turn forwards the input to the corresponding controller helper class. The helper class then interprets the input and turns it into a force that is applied to the model. Figure 2-3 shows the relationship between the classes of the Sensor-Input-Controller system.

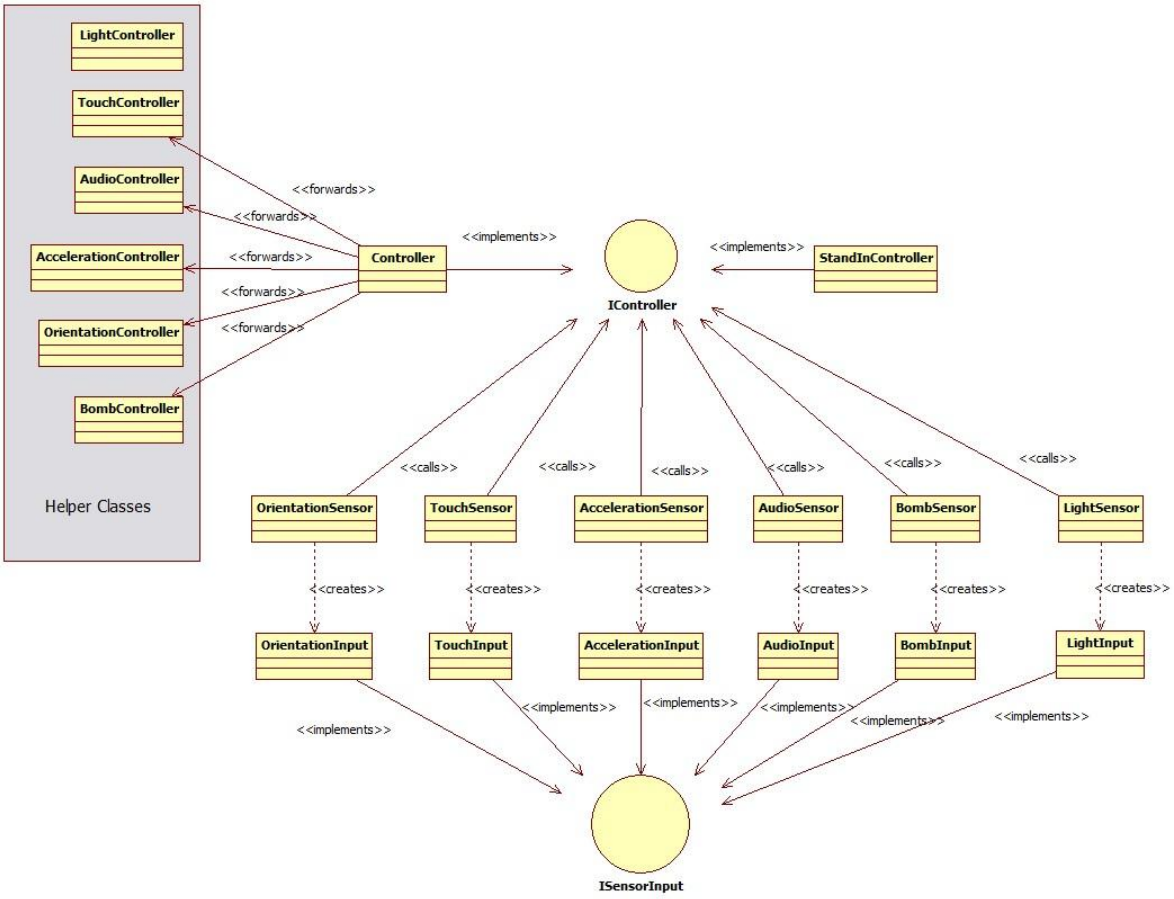

Figure 2-3 the Sensor-Input-Controller System

3. Interpreting Sensors

3.1. The Physics Engine

The *Model* manages a thread referred to as "the physics engine", which is responsible for applying forces to the balloon. Every 10ms the physics engine recalculates the position, speed and acceleration of the balloon. It can be roughly described in pseudo code as:

```
// update speed and position
newSpeed = oldSpeed+elapsedTime*acceleration.
newPosition = oldPosition + elapsedTime*((oldSpeed+newSpeed)/2)
// calculate current force
totalForce = 0
for(Force force : forces) {
      totalForce += force
}
// calculate acceleration
Acceleration = totalForce/mass
```

where `forces` is an array containing all the forces applied to the model in the last 10ms. All controller helpers that handle inputs apply the forces they calculated in the last 10ms to this array. In addition, the physics engine calculates a resistance force that also acts on the balloon. This resistance force is proportional to the square of the balloon's current speed:

```
fFriction = (1/Fr)*speed^2
```

where `Fr` is a fixed friction parameter. The reason for this resistance force is to prevent the balloon from gaining too much speed and becoming uncontrollable.

3.2. Sensor Readings

Death to Balloons uses both the acceleration as well as the orientation Sensor provided by Android's *SensorEvent* API. The following description of the *SensorEvent* API helps to understand how sensor readings are converted into control inputs:

Definition of the coordinate system used by the *SensorEvent* API (6)

The coordinate-system is defined relative to the screen of the phone in its default orientation. The x-axis is horizontal and points to the right, the y-axis is vertical and points "up" and the z-axis points towards the outside of the front face of the screen.

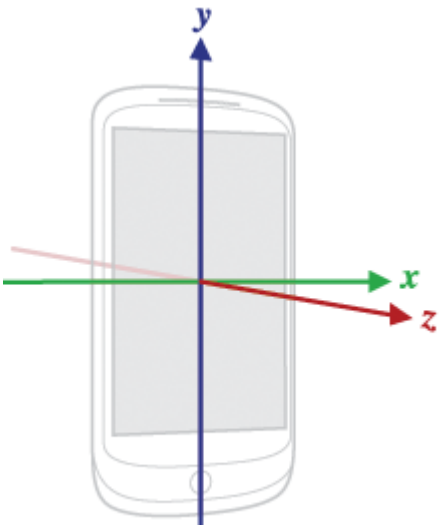

Figure 3-1 The Coordinate System (6)

Orientation Sensor (6)

The orientation of the phone is described by an orientation vector. This vector consists of three values:

1. The azimuth, which is the angle between the magnetic north direction and the y-axis with respect to rotation around the z-axis.
2. The pitch, which is the rotation around the x-axis
3. The roll, which is the rotation around the y-axis.

Acceleration Sensor (6)

This sensor measures the acceleration applied to the device. The acceleration is a vector of three values describing the acceleration in the direction of each of the axes. It has the units m/s^2.

LightSensor (6)

This sensor measures the ambient light level in lux.

3.3. Control Schemes: From Sensor Readings to Forces

In this section we cover how the different control schemes work in detail.

3.3.1. The Touch Control Scheme

When a user touches the screen, any class that is registered as an *OnTouchListener* will receive motion events. A motion event includes quite a bit of data. The game implementation only uses the x and y coordinates specifying where the user touched the screen. The force is simply calculated as:

```
f = (positionOfBalloon – positionOfTouchEvent)*factor
```

Performance

This control scheme delivers very exact controls, but such controls are standard in many games and so do not deliver any new experience to users.

3.3.2. The Jolt Control Scheme

The jolt control scheme uses the acceleration sensor to discover in which direction a user moves his phone. A jolt is a rapid movement consisting of an acceleration-deceleration phase as the user moves the phone in the direction he wishes to move the balloon followed by an acceleration-deceleration phase as the user returns the phone to its original position.

As a result, the difficulty in getting this scheme to work is to correctly discover when the user begins his motion and to stop reading values before he reverses the direction of movement.

To achieve this we apply the following strategy:

1. Wait for the phone to start moving. This is done by comparing the current acceleration to the one measured previously. Once the difference between the two is above a threshold we start recording acceleration values.
2. We continue to record values for 90ms, which will record several values.
3. Depending on whether or not the largest value measured comes before the smallest value, we choose the acceleration to be in one direction or the other.
4. After we stop recording values we do not start listening again for 350ms. This “cool down” period prevents measurements during the rest of the movement being wrongly interpreted as a new jolt.

The force applied to the model is then simply:

```
F = acceleration*parameter
```

The 90ms are quite short, and, thus, a major limiting factor to the accuracy of our prediction is the speed at which the sensor can sample. Android’s *SensorEvent* API does not specify how long the delay is between values. However, measurements by the community indicate that at the quickest sampling rate the sensor updates on average every 21ms with a standard deviation of 8ms (7).

This means that, in 90ms, we can only count on 3 to 4 values being read. It is very possible that these will not be enough to correctly determine the direction. Furthermore, the cool down period is problematic. What if the user takes longer than the 350 ms duration to finish his gesture? What if he attempts another movement before cool down is over, resulting in values being read during the middle of duration which will lead to an incorrect result? These problems were out of the scope of the project.

Performance

Considering the issues mentioned above it is perhaps not of much surprise that the performance of this control scheme is rather poor. While it is indeed possible to detect the direction of the movement reasonably accurately, the speed at which the gesture is performed is far more difficult to measure. Changing the duration of the various stages does change the accuracy, but experimental results indicate that 90ms for the reading duration and 350ms for the cool down period deliver the best, but still limited, accuracy.

Due to the limited accuracy of this control scheme, the user can find it frustrating at times.

3.3.3. The Air Bubble Control Scheme

This control uses the roll value to determine the force acting on the balloon in the x direction the pitch value to determine the force acting on the balloon in the y direction. Both the roll and the pitch measurements are precise and noiseless, thus calculating the force applied to the model is simply a matter of multiplying the values read from the sensor by a parameter.

Performance

Like the touch control, this control scheme is accurate but rather simple.

3.3.4. The Wind Compass Control Scheme

This control scheme uses a combination of microphone and orientation sensor readings to simulate wind. The intensity of the wind is determined by how loud the user makes a “shhh” sound.

To determine if a user is making a “shhh” sound, we use the zero crossing method, which counts the number of times the recorded values change from positive to negative and vice versa. This allows us to estimate the frequency. The phone is set to sample at 8000Hz and the encoding is 16-bit PCM.

The pseudo code which achieves this is:

```
while(true)
{
	numCrossings = 0; //initialize the number of zero crossings to zero
	for (p = 0; p < bufferSize - 1; p++) {
		if (audioData[p] > 0 && audioData[p + 1] <= 0)
			numCrossing++;

		if (audioData[p] < 0 && audioData[p + 1] >= 0)
			numCrossing++;

		avgVolume += Math.abs(audioData[p]);
	}

	avgVolume += avgVolume/bufferSize;
	avgCrossing = avgCrossing + numCrossing;

	if(elapsed > 10ms)
	{
	 // calculate average number of crossings and volume in the last
	 // 10 ms
	}
}
```

As it turns out, this simple process delivers surprisingly useful results. The closer the sound is to white noise the higher the number of zero crossings. The human “shhh” sound consistently results in zero crossing readings of over 500 compared to any other sound. Choosing this sound as the sound for wind allows us to screen out all other noises such as talking.

To determine the direction in which the wind blows, the wind compass control scheme uses the azimuth value delivered by the orientation sensor. The idea is that, when the user rotates the phone around the z-axis and then blows, the balloon will move in the direction he is blowing across the screen. Because azimuth is measured with respect to the magnetic north pole, we calibrate the direction by measuring an initial azimuth value when the game is set up. Afterwards the angle at which the wind blows is calculated with respect to this originally measured value as follows:

```
relativAzimuth = currentAzimut + (360 – originallyMeasuredAzimuth)
```

The magnitude of the force applied to the model is determined by the volume, and the direction is determined by the relative azimuth.

Performance

Detecting the “shhh” sound works very well, as does calculating the magnitude from the volume. The azimuth values, however, can cause problems, even if the user does not move after measuring the initial azimuth. For one, the azimuth measurements are plagued by noise. Furthermore, in the presence of disturbances in the ambient magnetic field, the azimuth values can vary substantially. In this case, even small movements made while playing the game can greatly affect the apparent direction in which the wind blows, a situation that can be very unintuitive for users.

3.3.5. The Wind Roll Control Scheme

This control scheme attempts to correct the imprecision of the Wind Compass scheme by using the roll value of the orientation vector instead of the rather inaccurate azimuth value. This is the only difference between the Wind Roll and the Wind Compass control schemes.

Performance

While this combination of sensors delivers a control scheme which is very stable and consistent, it is not only far less intuitive then the Wind Compass control scheme, but also very awkward to use.

3.3.6. The Bomb Control Scheme

The Bomb Control scheme uses the touch screen to place bombs on the screen and then uses the acceleration sensor to discover when the users shakes the phone to blow them up. Discovering shaking is far simpler than trying to interpret in which direction the phone is moving. All it takes to discover shaking is to compare the previous acceleration value to the current value. If the difference between the two is above a certain threshold, the user is shaking the phone.

The force applied by each bomb drops exponentially the further the bomb is from the balloon:

```
Force = parameter*exp(-a*distance)
```

The reason for this mathematical model is simply that it works well in the game and not because it's an accurate simulation of a real explosion.

Furthermore, once a round the survivor can erase the placed balloons by covering the light sensor with his hand. This is implemented by having the light sensor store a reference value at the beginning of each round. If the light drops below three quarters of this reference value the sensor records a detection.

Performance

Placing and detonating the bombs works very well and is an excellent example how even a simple use of the sensors provides a more intuitive and ultimately more fun control scheme. The light detection works well, as long as the ambient lighting is reasonably bright and does not change during a round. There are, however, two issues. The first is that the light sensor is quite slow to respond, taking close to a second at times, which means a user cannot quickly erase the bombs. Second, it is not obvious where the light sensor is located, in particular, on the Nexus 1, it is located on the top left of the phone. Without this knowledge a user may be confused.

4. What could be improved

4.1. Improving Interpretation of the Azimuth Sensor Readings

The sensor readings from the azimuth are rather poor and difficult to deal with. There are three difficulties which need to be tackled before this control scheme can be considered solid:

1. The first difficulty is that the azimuth sensor is noisy, even if the phone is stationary. A good filter would be required to filter this noise.
2. The second and far more difficult problem is dealing with disturbances in the ambient magnetic field. A solution would require knowledge of the ambient field in order to compensate for movement within this field.
3. The third difficulty is figuring out what to do if the user changes position or orientation. In this case the relative azimuth will be calculated incorrectly. There should be a way to reset the initially measured azimuth during the game.

4.2. Improving the Jolt Control Scheme

The jolt control scheme is flawed. Simply trying to discover when a user begins his movement and then measuring a few values in an attempt to discover the direction the user is moving the phone, is not sufficient. It may be possible to take measurements for a longer period and then discover the movement the player made by analyzing this larger set of data. However, this is problematic because it increases the already long duration between control inputs. This difficulty along with the fact that updates for the acceleration vector come only every 21ms means that improving this control scheme would require serious modification to make it work. The algorithm we used is clearly not good enough. Perhaps a combination of “very good” filters combined with heuristics based on pattern matching would be successful. In the worst case it may require better sensors.

4.3. Support for more than two Players

The next step in expanding the game would be supporting more than two players. The software already has many features that could support multiple players and multiple balloons. For example, balloons are stored in an array, as are players. However, the classes dealing with the connection between phones would need to be rewritten, as one needs to add control logic that can manage multiple connections. Another issue that needs to be addressed is the additional contention for shared resources that comes with having more players. This means there would be more frequent access to synchronized objects. Due to the fact that acquiring and releasing locks can be very resource intensive, a game with more than two players may experience lag.

4.4. Testing the Game on different Hardware

The target device for this game is the HTC Nexus 1, and it was with this model that all testing was performed. However, there are many different phone models on the market running the Android operating system. How well the game performs on these platforms is unknown.

4.5. Balancing Control Schemes

The various control schemes implemented in this project differ greatly in the way they calculate the forces applied to the balloon. There are various parameters that influence the forces and that were determined experimentally. Whenever a new control scheme was implemented, some time was spent adjusting the parameters so that the control “felt” good. However, apart from the Wind Compass vs. the Bomb control scheme, very little time was spent testing play in which the players

used different control schemes. As a result it is very likely that certain control schemes have an inherent advantage over others.

4.6. Add more Control Schemes

The game is written such that adding more control schemes is quite easy. One can think of many control schemes that were not implemented. Some examples we came up with are:

4.6.1. Fire Arrows at the Balloon

A player using these controls would have an arrow which he can move along the side of the phone. This could be achieved just like in the Wind Compass or the Wind Roll schemes, by using the azimuth or roll value delivered by orientation sensor, respectively. To fire the arrow the user would tap or shake the phone. This could be detected using the acceleration sensor.

4.6.2. Use Portals to teleport the Balloon

In this control scheme the player could draw a portal by touching the screen with his finger. He could then teleport the balloon to the location of the portal by tapping the phone. The direction, speed and acceleration of the balloon would be conserved when the balloon teleports.

4.6.3. Detecting Noises

While the simple method of counting zero crossings in this project is insufficient to interpret speech, it could possibly detect simple noises. For example, clapping one's hands results in the number of zero crossings rising and then dropping quickly. The same is true for clicking one's tongue. This could be used to move the balloon. The difficulty with such a control scheme is that it may be affected by ambient noise.

4.7. Add more Features

It may be interesting to add more gameplay features. For example one could introduce "power-ups", items that change the rules of the playing field when they are activated. For example, power-ups could increase the speed of the balloon, add more balloons or force a change of the control scheme. The possibilities for enhancing gameplay and making interaction with the balloon more exciting and challenging are limited only by our imagination.

The difficulty in implementing such power-ups, which dynamically change the gameplay, is that it would require a significant refactoring of the code because many of the gameplay rules are hard-coded into the *Model* class.

5. Conclusion

Death to Balloons is a simple multiplayer game for two players on the Android platform in which players take turns attempting to pop or protect a balloon. Communication between the two phones is achieved through a Bluetooth link. The game allows the players to choose from a variety of control schemes, from rather traditional input methods that use the touch screen, to more exotic control schemes such as the Wind Compass controls where players blow into the phone, simulating a wind acting in the game. Death to Balloons shows that it is possible to take advantage of the many sensors available to smartphone developers and to create games using control schemes that have not been thought of before.

An enjoyable game experience requires that user input be reliably accurate and repeatable. It is no fun when the object under control acts differently than expected, or when the same input action results in an inaccurate, or even apparently random, response. As we discovered in this project, delivering such an experience is not always easy. For example, the inherent hardware and software challenges encountered in the jolt scheme, where the inaccurate values delivered by the acceleration sensor resulted in a frustrating game, illustrate the difficulties one can encounter. In contrast, the bomb scheme shows that it is still possible to integrate the acceleration sensor into a precise and fun method of input by having it trigger bombs placed by one's finger, thus indirectly affecting the balloon. Such out of the box thinking can result in ways of interacting in the game that are refreshingly new, yet still very exact.

We conclude by saying that it is worth thinking hard about how to use the many sensors of modern smartphones when creating new control schemes for games. This is still widely uncharted territory, and a clever control scheme may set your game apart from the ferocious competition in the mobile video game industry.

Bibliography

1. Mobile Gaming Market Tops $800 Million in 2010. *emarkter.* [Online] August 18, 2010. http://www.emarketer.com/Article.aspx?R=1007874.

2. Android Market. *SPACEWILLI - The incredible voice-controlled game APP - absolutely spacy...* [Online] https://market.android.com/details?id=de.hoch3.spacewilli.

3. **Saint, Nick.** Business Insider. *The 20 Best Selling Android Games Of All Time.* [Online] 11 9, 2010. http://www.businessinsider.com/best-android-games-2010-11#.

4. **IGN.** IGN. *Best Mobile Games.* [Online] http://bestof.ign.com/2010/mobile/index.html.

5. **Eriksen, Erik Holthe.** *MIT Entrepreneurship review.* [Online] http://miter.mit.edu/article/angry-birds-will-be-bigger-mickey-mouse-and-mario-there-success-formula-apps.

6. **Google.** Android Developers. *SensorEvent.* [Online] http://developer.android.com/reference/android/hardware/SensorEvent.html.

7. **Frank, Jordan.** *Accelerometer frequency.* [Online] 2 2009. http://osdir.com/ml/AndroidDevelopers/2009-02/msg03111.html.

8. **Google.** *Android Developers.* [Online] http://developer.android.com/guide/basics/what-is-android.html.

9. The picture used as the background in Death to Balloons is an adaptation of the photo “Snow” by Aleksander Ishere, available under a Creative Commons Attribution-NonCommercial license. The original picture can be found at: http://www.flickr.com/photos/46422886@N06/4263132148/