Android practical training

Winter term 2011

Baris Oztop, Stefan Matl

**Documentation: 3D Game App**

Mentoring:

Nils Kannengießer

TUM, Fakultät für Informatik

Lehrstuhl 13, Prof. Dr. Uwe Baumgarten

Boltzmannstr. 3

85748 Garching bei München

Contents

[Abstract 2](#_Toc315821624)

[Project Team 3](#_Toc315821625)

[Baris Oztop 3](#_Toc315821626)

[Stefan Matl 3](#_Toc315821627)

[Project Plan 3](#_Toc315821628)

[Idea 3](#_Toc315821629)

[Product 3](#_Toc315821630)

[How to play 3](#_Toc315821631)

[Differences 4](#_Toc315821632)

[Quality requirements 4](#_Toc315821633)

[Must-have features 4](#_Toc315821634)

[Optional features 5](#_Toc315821635)

[Implementation 6](#_Toc315821636)

[Resources 6](#_Toc315821637)

[GUI 6](#_Toc315821638)

[Graphics 7](#_Toc315821639)

[AI 7](#_Toc315821640)

[Basic interfaces and classes 7](#_Toc315821641)

[Project schedule 8](#_Toc315821642)

[Project Flow 8](#_Toc315821643)

[Detailed Description 9](#_Toc315821644)

[Bluetooth Communication 9](#_Toc315821645)

[Classes 9](#_Toc315821646)

[MultiplayerActivity 9](#_Toc315821647)

[ModeSelectionActivity 9](#_Toc315821648)

[ClientNumberPicker 9](#_Toc315821649)

[DeviceListActivity 9](#_Toc315821650)

[BluetoothMPService 10](#_Toc315821651)

[DataServer / DataClient 10](#_Toc315821652)

[Server and Client Side Bluetooth Communication 11](#_Toc315821653)

[Server Side 11](#_Toc315821654)

[Client Side 14](#_Toc315821655)

[Communication with the Main Activity 15](#_Toc315821656)

[Actions According to the Activity Results 17](#_Toc315821657)

[Detailed Test Plan 18](#_Toc315821658)

[Devices that we tested on 18](#_Toc315821659)

[Testing different screen resolution and size 19](#_Toc315821660)

[Testing Multiplayer Mode 19](#_Toc315821661)

[Project Conclusion 19](#_Toc315821662)

# Abstract

# Project Team

## Baris Oztop

Baris is currently Informatics Master student at TU München, and this is his second semester. He is graduated from Middle East Technical University, Turkey with Computer Engineering major. His favorite object-oriented programming language is Java. He has done several projects in his bachelor and first semester of his master degree. Some of these projects are Database Implementation, Information Visualization, 3D User Interfaces, Rapid Application Development, and Peer-to-Peer Desktop Search Agent. He is currently working as student trainee in Software Development and Consultant Company. His hobbies are Movie and Music Collecting, Photography, Travelling, World Cuisines, and Cooking.

## Stefan Matl

# Project Plan

## Idea

‘Mensch ärgere dich nicht’ is a strategy board game for four players in the classic version. The name of the game means ‘Do not get angry’. The game is one of the famous classical games for all the age groups. It is a good alternative to turn the people’s waiting time e.g. in subway into joyful moments with their peers. We developed the game to get the advantage of the Android devices Bluetooth technology. Hence, it is possible to connect 8 mobile devices to each other to enjoy the game. While 4 of those devices’ users playing the game on their devices, the other 4 player can watch the game on their devices’ screen.

## Product

The product is provided to support Multilanguage. It comes with predefined languages for English and German. The application is designed for Android systems with Android 2.1 or higher. It requires an API 7 or higher. However, Android systems version 2.3.4 and below don’t support more than 2 devices connecting to another one via Bluetooth. Therefore, if users would like to play the game with total 8 devices, they have to make it sure that the server device has the Android OS version 2.3.5 or newer.

### How to play

Each player (possible number of players: two, three or four players) has four pegs and aims at getting them from their first position, out-region, into their final home fields. First, the player has to throw a dice. In the beginning all the pegs of the player are blocked and the player is allowed to throw the dice three times until throwing a six. With every six, a player can unblock one peg from its out-region, and in each round the peg is moved according to the number the dice shows. A player is not allowed to move the peg on a field already occupied by one of his other pegs. If the field is occupied with another player’s peg, he can kick out the other player and force him to restart with that peg.

### Differences

Basically the application is intended to work and behave the same way a real board game would react. But in contrast to a real game, where people are sitting around a table, it is not possible to manipulate the dice or influence the number the dice shows. Once the player has thrown the dice, he has to move his selected peg by exactly that given amount and nothing else. It is not possible to cheat or move the pegs of another player. A move cannot be undone. Application’s GUI allows user to rotate the board, and zoom in and out to the board.

## Quality requirements

The app will neither crash during an appropriate usage nor while a time-out or not-responding exception is sent. If there is a connection problem during the game, other device users are notified via a toast message including the problematic device name in it.

### Must-have features

The following features are non-optional and therefore cannot be removed or skipped:

* *Resources like pictures, models and sound effects:*

In order to make the game more interesting for the player, the graphics and models have to be provided in high detail. Resources like textures are needed for the rendering.

* *Graphical user interface for controlling the game:*

The GUI allows the player to control the game. With the GUI the player can select and move his pegs on the board or change game settings.

* *3D game rendering:*

With a 3D rendering the game should look and feel like 3D. With OpenGL ES we can define properties like perspective or depth for the 3D rendering. Therefore all the game objects, like pegs, board or dice should be rendered in 3D.

* *Enabling connection with up to three other mobile android devices:*

In order to allow multi-player game mode, the app has to connect to multiple different devices where the same app is already installed. After setting up a Bluetooth connection, two or more players can join the same game and play against each other.

* *At least one artificial intelligence in order to enable a two-player mode:*

If there is only one single player and no one else available, the app has to start an artificial intelligence, meaning a computer-based non-human player acting as an enemy for the player. In that mode no connection is needed.

* Hurry-up mode:

If a player doesn’t move within a given amount of time, the app will make a move for him, which is either randomly selected or a calculated worst-case move. In case a player takes too long many times, his best peg is sent back to the start position. The user can set this maximum waiting time in the settings or he can turn this mode off.

### Optional features

Optional features will be implemented if there is time left after the implementation of the must-have features:

* *Real-time rendering:*

This defines a special type of rendering where all the objects are rendered fast enough, so that the player doesn’t notice any time delay or other kinds of interruption. The rendering itself should be separated from the game calculation.

* *Connection for up to eight players:*

There are different versions of the game, e.g. one with an eight-player mode. In order to realize that, the app requires an option to connect with seven other phones.

* *Multiple artificial behaviors in order to increase game fun:*

Because only one AI is easy to beat, there have to be multiple different kinds of computer-based players. Each of them needs another algorithm for calculating the next move. It might even be possible to implement them in a way to provide them on different levels like ‘easy’, ‘medium’ and ‘hard’.

* *Animations like zooming the camera or moving pegs:*

In order to increase game fun, it would be a nice feature to play some animations and have some special effects, e.g. when a peg is moving or forced to return home or a player wins the game.

* *Shaking feature for supporting a realistic dice throwing:*

Because random methods are sometimes not very efficient and don't really return a random number, it would be a great feature and of course providing additional fun, if the player could shake the phone in order to throw the dice. Based on the movement of the player a number is calculated.

* *Dynamic moves:*

Whenever a player makes a move, an animation shows his peg sliding from one field to another. By providing a high frame rate, the single steps will look like one dynamic move.

* *Support of different languages:*

In order to increase the amount of possible customers, the layout of the app is given in multiple different languages, e.g. German, Spanish, French or Chinese.

## Implementation

The app was implemented with Eclipse Version: 3.7.1.r37 with ADT Plugin for Eclipse Version 16.0.1. The Android platform that we used was Android 2.2 (API 8). Although the latest version of the Android platform was 4.0 (API 14) during our development, we preferred to use 2.2 to support old Android devices as well. Java version was 1.6.0\_20, and development platform was 32 bit.

### Resources

* *Pictures:*

In order to reduce calculations, some basic layouts can be drawn with programs like GIMP or other drawing software. Also the app needs an icon, so that everyone can identify it, if it is available in the Android market and for a better marketing. Parts of the models or the board can be textured with images.

* *Models:*

For a 3D rendering, 3D models are needed. Models can be used in order to render a peg, the game board and the cube. This 3D data can be created with software like SolidEdge, SolidWorks, Catia V5 or any other modeling software. Even small Java programs could be used for calculating 3D models.

* *Sound:*

During special effects e.g. when a peg is sent back or a player wins, it is recommended to play some sound. If there is enough time to implement this, a sound will be played all the time during the game. Therefore the player should have an option to turn it off.

* *Path on the board:*

Every single peg is only allowed to move along special paths on the board. Those paths can be implemented as a graph or an array.

### GUI

A graphical user interface is required in order to control the game. Therefore touch events have to be registered as well as a change in gravity or acceleration if the player shakes the phone. With the use of an OnTouchListener and a SensorListener it is possible to catch those events and process them. The GUI will provide an option menu to change some game settings or enable the user to start and quit the game. Most of the screen will be used to show the game and has to be updated very fast, meaning a very high frame rate. By touching the screen, the player can e.g. select his current peg which he wants to move. The game has to match the touch coordinates with the 3D coordinates in order to select the right peg. Also the player can move the camera and zoom close to his pegs. Of course the GUI shows the player the current situation, meaning that after every move all the players (devices) have to receive an update.

### 

### Graphics

The game can be rendered with OpenGL ES, which can be used as a package, which provides high-level methods in order to paint the single models. Those need to be read and converted from a 3D resource into a triangle-based model. Once it is written to a byte buffer, OpenGL will add it to an intern 3D matrix. Additional parameters like luminosity and depth can be set.

### AI

In order to implement an artificial intelligence it is recommended to analyse the game as well as different strategies and choose the best ones for the behavior of the AI. For an algorithm that calculates the next move it is important to have certain criteria to find and rate a move. Usually the AI can calculate moves during the player’s turn, meaning that the calculation can be done parallel to the rest of the game. If different AIs are implemented (see 4.2 Optional Features), the user can set the level of difficulty of the current one(s).

### Basic interfaces and classes

classes: PLAYER, DICE, PEG, BOARD, GLSURFACEVIEW

interfaces: RENDERABLE

PLAYER:

properties: name, color

methods: dice()

PEG extends RENDERABLE:

properties: number, color, position

methods: move(), render(), return\_home()

BOARD extends RENDERABLE:

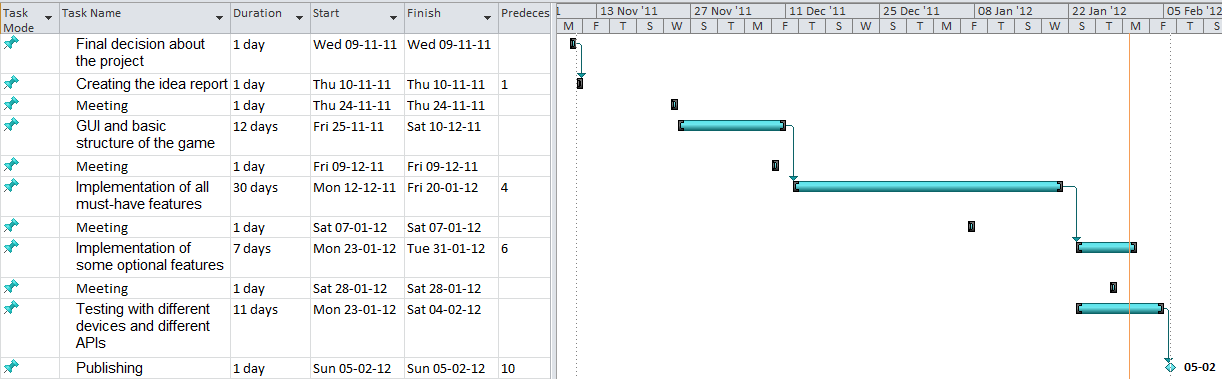
properties: size, paths, color, texture

methods: render()

RENDERABLE:

methods: render()

## Project schedule



# Project Flow

Below you can find the problems that we have encountered during the development phase of the game application:

* We wanted to develop an application to run on the earlier versions of Android OS to make our game accessible by more people. Therefore we used the API level 10 (Android 2.1). However, there was a case where standard methods, which is also written in the Android developers’ page, didn’t work. One of these cases was showing a progress bar while client devices are connection to server device. Hence, server device user will know how many users are connected via the completion level of the progress bar. We initialized the progress bar object with the corresponding activity’s context by getting it with getApplicationContext() method, however this didn’t work. This was a hard to spot failure that we spent nearly one and a half day to figure what the problem is. After our research, we figured out that Android 2.1’s progress bar initialization doesn’t get the Activity’s context with getApplicationContext()method but simply calling the *ActivityName.this* technique. However, getApplicationContext() works only newer versions of Android. This is the case just for initializaing the progress bar, the other calls for getApplicationContext()still works in Android 2.1.
* We wanted our game application able to handle 7 client devices connected to a server device. Therefore, while 4 devices’ users are playing the game, the other 4 devices’ users might be able to watch the game with their smart phones. However, during our tests we realized that the third and more devices are not able to connect to server device whose Android version is 2.3.4 or below. This was the most time consuming problem that we encountered during the development of the Bluetooth communication.

# Detailed Description

## Rendering



In order to make OpenGL easy to use we separated the graphical rendering from the 3D-modelling architecture. First of all, we define an interface called renderable including the method render. That means every implementation of this interface will have to provide a basic rendering function. For developing a renderable into a geometric object, we define an abstract concept, that requests more methods like transfer the position, or for changing the visibility, because a geometry can be visible or not. Our first not abstract class, that implements the render-method, is a simple geometric object. That means, we want to wrap simple geometry to the OpenGL ES concept. Two examples, we will use later, are the triangle fan and the triangle stripe. Those constructs are types, that OpenGl can use for rendering as well as some GPUs. Additional to the type of geometry, we have to add certain values to define the color, textures and of course the 3D-coordinates (mesh) of our object. Therefor our simple geometric object saves three buffers, which contain the data. Triangle fan and stripe have a constructor only, because they just define the correct OpenGL native type, so that the user doesn't have to deal with that. All other values are just forwarded to our simple geometric object, which is in charge of rendering.

Creating a big game just with triangle fans and stripes is tiresome, so we define a game object, that stores a group of simple geometric objects (see sgobjects) and represents a single 3D-model in the game, like the dice, the board or a peg. The game object is not responsible for creating the object, but for managing them. The game object is on the same level with the simple geometric object and so it also defines the renderable method render, which is used for calling the stored geometric objects to render. As you can see, it also knows the current position that is defined by x, y and z. This concept of storing the position in the game object and not in the mesh, will be very important for saving memory and improving rendering performance due to some problems with moving objects.

Before talking about memory and improving the code, let's see how the actual rendering of the simple geometric object is done. Here we have to know that the calculation of the rendering, meaning the calculation of vertices, colors and more, is already done by native OpenGL. OpenGL ES just wraps a bunch of methods for easy use in Java. So all we have to do, is sending the data to the OpenGL, of course besides some additional settings. To show the rendering we use a view, that is just one component on the phone's screen. We decided to use a GLSurfaceView with is a special view for rendering with OpenGL. Then we add a renderer to that view. This renderer has basically three important methods, so the first thing to do is implementing our own renderer.

**public** **class** GameRenderer **implements** android.opengl.GLSurfaceView.Renderer {

**public** **void** onSurfaceCreated(GL10 gl, EGLConfig config) {}

**public** **void** onDrawFrame(GL10 gl) {

// rendering

}

**public** **void** onSurfaceChanged(GL10 gl, **int** width, **int** height) {

// basic settings

}

}

OnSurfaceCreated is called just once, when the surface is created. Currently we don't need it. OnDrawFrame is called automatically from the surface view, with a high frame rate. That's where we do our rendering. OnSurfaceChanged is called, when the surface changes. This is the case, when e.g. the phone is rotated and we get a new surface. It is also called for the first time, after creation. Here we get our width and height as well as the OpenGl object (GL10) for rendering. The name GL10 just comes from it's version. Beside saving the view dimensions, we set some basics, like the viewport for OpenGl, which contains the view dimension. With

gl.glMatrixMode(GL10.*GL\_MODELVIEW*);

gl.glEnable(GL10.*GL\_DEPTH\_TEST*);

gl.glEnable(GL10.*GL\_COLOR\_MATERIAL*);

we set the matrix mode to model view. That means the matrix which is used intern for rendering is now adjusted for a typical 3D-view. E.g. we could also use GL10.GL\_PROJECTION instead, but than we had another perspective. Then we enable the depth test, which means, that OpenGl is testing the position of an object before drawing it. If there already is another object in front of it, it won't be drawn. Next line enables color material. We need that to get colorful pictures instead of gray ones. Now OpenGL will deal with our given colors.

We already mentioned that we have to send the data to OpenGL. Actually sending might be the wrong word, because this would cause a lot of performance and memory waste. So we just give some kind of pointer to OpenGL. In order to make the data compact, we use a direct byte buffer.

ByteBuffer buffer = ByteBuffer.*allocateDirect*(SIZE = VALUES \* 4);

buffer.order(ByteOrder.*nativeOrder*());

FloatBuffer bufferFloat = bufferByte.asFloatBuffer();

We create a float buffer from the byte buffer, where we specify the size. Because it is a byte buffer and we want to put float values, we have to multiply our values' amount with four (or eight, depending on the operating system). We can get the float size with 'Float.*SIZE* >> 3'. Then we specify the byte order and create our float buffer. With put(**float**[] values) and clear() , we can add and remove our values in the buffer. With

gl.glEnableClientState(GL10.*GL\_VERTEX\_ARRAY*);

gl.glEnableClientState(GL10.*GL\_TEXTURE\_COORD\_ARRAY*);

OpenGL will accept those buffers for sending vertex arrays and texture coordinates. We will enable more buffers later (e.g. colors), but those we use all the time and that's why we add them in the onSurfaceChanged-method.

Now we can start with the rendering process, that is done for each frame in onDrawFrame. Because OpenGL not just renders our model, but the our frame with it's background as well, we have to take care of that, too. First we clear the background and make it black. Then we clear the intern color and depth buffers of OpenGL.

gl.glClearColor(0, 0, 0, 0);

gl.glClear(GL10.*GL\_COLOR\_BUFFER\_BIT* | GL10.*GL\_DEPTH\_BUFFER\_BIT*);

Then we have to set the perspective. Therefor we use the GLU-library. We simple put the GL10, the screen relation and aspect, width, height and two distances. The first distance is the minimum distance between the virtual camera and any object, that will be rendered. The second one is the maximum. Any objects further than our maximum are not rendered anymore. The next line sets the camera. We put it's center, the point we look at (0, 0, 0) and the up-vector (0, 0, 1), which specifies the rotation of the view.

GLU.*gluPerspective*(gl, 67, *width* / *height*, *distance\_min*, *distance\_max*);

GLU.*gluLookAt*(gl, *center\_x*, *center\_y*, *center\_z*, 0, 0, 0, 0, 0, 1);

Room.*render*(gl);

After that we can call all objects for rendering. As you can see in the last diagram, we also have a room for adding renderables (simple geometric objects as well as game objects) and calling all for rendering. Because there will be only one room and also we don't want to spam with room references, everything in room is static and synchronized to the room class. By changing the camera values, we can also make the game rotation.

Our call for rendering starts the hole rendering process of each single object. The simple geometric objects, which have the buffers for coordinates, colors and textures, will render them by sending them to OpenGL. So let's look at the render-method of the simple geometric object.

/\*\* {@inheritDoc} \*/

**public** **final** **void** render(GL10 gl) {

// checking whether current object is visible

**if** (!visible)

**return**;

bufferV.rewind();

gl.glVertexPointer(3, GL10.*GL\_FLOAT*, 0, bufferV);

**if** (bufferC != **null**) {

// enabling a color array

gl.glEnableClientState(GL10.*GL\_COLOR\_ARRAY*);

// setting the position to 0

bufferC.rewind();

gl.glColorPointer(4, GL10.*GL\_FLOAT*, 0, bufferC);

} **else** {

// disabling a color array

gl.glDisableClientState(GL10.*GL\_COLOR\_ARRAY*);

**if** (bufferT == **null**) // simple color

gl.glColor4f(color[0], color[1], color[2], color[3]);

**else** { // textures

bufferT.rewind();

gl.glEnable(GL10.*GL\_TEXTURE\_2D*); // enable textures

// set current texture gl.glBindTexture(GL10.*GL\_TEXTURE\_2D*, Textures.*getId*(texture));

gl.glTexCoordPointer(2, GL10.*GL\_SHORT*, 0, bufferT);

}

}

// rendering this object

gl.glDrawArrays(type, 0, amount);

gl.glDisable(GL10.*GL\_TEXTURE\_2D*);

}

After verifying, whether the current object is visible, we start putting the vertices to openGL. They are in bufferV. We already enabled the vertex buffer client state, so we can just set the reference (pointer) to OpenGL. Then we have to see, whether we have a color buffer or not, because there are many ways to specify the colors for OpenGL. If our bufferC is not null we enable the color buffer client state and put the color buffer there. Like with the vertices, we have to rewind the buffer in order to set the start position for reading to zero. Also we define the way our colors are stored in the buffer. Here we have four values for the alpha, red, green and blue part of the color. The type is float and the offset to the next color is zero. The same way is used for vertices. The first argument for the vertex pointer, which is three, just defines, that our vertices have three dimensions (3D-model). Back to the color. If the buffer is null, we can either use another way for defining the colors or we use textures instead. So we disable the color client state. If we don't have a texture buffer, we set our colors with glColor4f. That means, that we don't specify a single color for each vertex (this way we would use the buffer), but define one color for the hole mesh (that is in the current vertex buffer). If we have textures, we won't need color. First we enable 2D-textures, then we specify which texture we want to use and put the texture buffer to OpenGL. We have to bind the texture, because OpenGL always refers to the current state. So when we enable a buffer or a texture id, then those ones are used, until we disable or change them. The texture buffer does not contain the texture itself, but additional values to describe the position and size of a texture (typically a bitmap) on the 2D-surface (e.g. a triangle). Let's finish with the rendering, before we look at textures. Ee just call glDrawArrays, which means that all currently set buffers and ids are rendered. Here we set the type, e.g. GL10.*GL\_TRIANGLE\_STRIP or GL10.GL\_TRIANGLE\_FAN.* Again our offset is zero and the amount says how many vertices are rendered. Last thing is to disable the 2D-textures. That's all. Our object is rendered !

Now let's take a quick look at textures. You maybe realized the textures class in the previous diagram. It is just for managing textures, because when we create our objects at game setup, we don't have the OpenGL, yet. So we store bitmaps (our textures) in a static list (Same concept like with the room). Later on, when we get our GL10, we bind all textures to an id. The simple geometric objects use another id they got, when the bitmap was added to the list. A good moment to bind the textures is when the surface is created. So we finally have a reason to use that method. By the way or surface doesn't change, so it doesn't matter, whether to do things in onSurfaceCreated or onSurfaceChanged. And that's how we bind the textures:

**public** **static** **final** **synchronized** **void** bindTextures(GL10 gl) {

**int** tmp[] = **new** **int**[*bitmaps*.size()];

// Generate texture-ID array

gl.glGenTextures(tmp.length, tmp, 0);

**for** (**int** i = 0; i < tmp.length; ++i) {

// Bind to texture ID

gl.glBindTexture(GL10.*GL\_TEXTURE\_2D*, tmp[i]);

// settings

gl.glTexParameterf(GL10.*GL\_TEXTURE\_2D*, GL10.*GL\_TEXTURE\_MIN\_FILTER*,

GL10.*GL\_NEAREST*);

gl.glTexParameterf(GL10.*GL\_TEXTURE\_2D*, GL10.*GL\_TEXTURE\_MAG\_FILTER*,

GL10.*GL\_NEAREST*);

gl.glTexParameterf(GL10.*GL\_TEXTURE\_2D*, GL10.*GL\_TEXTURE\_WRAP\_S*,

GL10.*GL\_REPEAT*);

gl.glTexParameterf(GL10.*GL\_TEXTURE\_2D*, GL10.*GL\_TEXTURE\_WRAP\_T*,

GL10.*GL\_REPEAT*);

gl.glTexEnvf(GL10.*GL\_TEXTURE\_ENV*, GL10.*GL\_TEXTURE\_ENV\_MODE*,

GL10.*GL\_REPLACE*);

// Build Texture from loaded bitmap for the currently-bound texture ID

GLUtils.*texImage2D*(GL10.*GL\_TEXTURE\_2D*, 0, *bitmaps*.get(i), 0);

}

}

When we get the GL10, we generate texture ids for each texture. Those ids are used for identification later during the rendering. Then we have a loop, where we bind each bitmap. First thing is to set the current id to OpenGL. Then we add settings, like how to render, how to scale or how to wrap. After that, we bind the bitmap to the texture with the current id. The texture class does a little bit more than that – we will see that later.

We already mentioned the camera and how to control it with a position, look-at and up-vector. In order to give the user many options for adjusting the view individual, we created a listener, that can call the renderer to change the view. This is part of our user interface and user interaction. Our game is totally 3D and we want to show that to the user as well. So the user can change the view, simple by touching the screen. Swiping over the screen means rotating the camera around the origin of our world, which is located in the middle of the board (position = (0, 0, 0)). If multi-touch is enabled, touching the screen with two fingers will lead to a zoom – either in our out. The camera is always located on half of a sphere around our world. The camera can move on lines along its surface and for zooming we just change the radius of that globe. So it's totally up to the user to play, watch and enjoy the game, whether in single game mode or not, from any position, that fits.

|  |  |
| --- | --- |
|  |  |
|  |  |



So now we are going to see more of the game object structure. Because the class GameObject still is an abstract one, we need to extend it with real game models. Game models are part of the rendering hierarchy, but they also interact as real game objects, meaning they behave according to their purpose. One important model is the dice. It provides methods for throwing it and it calculates a result. Like all game objects, it overrides the method action(), which is called from the render-method. That means in a way, that our game cycle is build on the rendering cycle. This is only possible and efficient, because we don't have any big calculations, that would cost too much performance. For example throwing the dice is one big animation, but calculating it's result is as easy as fast. There is always just one dice, so everything is static. There is a static dice object as well for using properties of the game object. So we have at least one dice object, for getting certain calls like for action or rotation. The dice has multiple methods for throwing – this will be explained later together with the game cycle.

Another game object is the board. Because we wanted it to be possible to create different boards, we defined a basic abstract class board, that requests lots of features, like a maximum amount of teams that can play on that board or the current playing teams (players). Of course it is possible to play with less players than the board is made for. It's subject to the board to create pegs. Also lots of game rules are done by the board. E.g. pegs can ask the board, if they are allowed to move or not. In order to stick to the project play, we designed the classic board, that is also known from the game product of Schmidt Spiele GmbH. That square consists of four equal formed regions, their only difference is their color. The board is made for four players with the colors red, yellow, green and blue. For each player there is a start region as well as a final row. Both consist of four square fields, matched to the player's color. For getting from the start fields to the final row, each peg has to cross all 40 fields, that make the path around the board. For determining the amount of fields to move, a player throws the dice. A draft of the board is shown below.



Because of the point symmetry, we only define coordinates for one quarter and then rotate it three times about 90 degrees and just change the color. On each single square, there will be a circle later. In order to paint the classic circles, we have to use textures, because OpenGl ES doesn't support any kind of round volumes or shapes, like circles, spheres or cylinders. Those methods, which particular exist in the native OpenGl code are just not wrapped with Java methods in this version. For increasing performance, we group as many squares as possible in triangle stripes. Please note, that the class diagrams above do not show every single property.

The two-step concept with an abstract basic class and the individual extension, that mainly defines the layout and not the behavior, is used for the peg, too. In the settings menu, the user can decide whether to use a simple peg, which is just some cuboid or the classic peg from the original game of Schmidt Spiele GmbH. A peg never knows it's absolute position on board (not the field number). It just has a value from 0 to 47, where 0 to 3 are the four start fields and 44 to 47 are the final fields. A peg has public methods for moving, resetting, checking whether it has already started on the board path, is still in the start position or has finished. Also there are functions for animations, like giveWay(dx, dy), which is used for telling the peg to move to the given side and back, so that another peg can pass. In case that a peg is controlled by a human player and not by an artificial intelligence, there is also a method for selecting it and making it highlighted. Therefor another color buffer is used. Those data is saved again in the classic peg or the simple peg.

We already mentioned that our OpenGl ES doesn't support the sphere, we need for our classic peg. So we have to create it out of triangle stripes. But we don't want to get visible stripes in the rendering, because, when light falls on a sphere the is always a color gradient, even if it is just the brightness. First of all, we don't want to use the light source, because of many reasons. One reason is decreasing performance. Second reason is that we would have to create all the vertex normals (not just face normals) and we need many lights, so that everything looks properly. Third, we don't want to have shadows on the board, because that might disturb the user. And fourth, we don't think we need light just for the classic peg models, because we can calculate that self and set an appropriate color buffer to our models. So we decided not to use lighting and adjust the peg's color instead.

However, we have triangles with different colors for the sphere. To avoid visible stripes from the color gradient, we don't set one color to each triangle, but many colors. In fact every vertex has it's own color. The big advantage is that OpenGl can use different colors for the vertices of a single polygon. The surface is filled accordingly to the colors and a gradient is created on each triangle. That's why you just see one round sphere and not the single objects. All in all for creating the sphere, we split it in multiple stripes horizontal and vertical. In the current version this are 20 and thirteen splits. Together with the round bottom of the peg, we have about 600 triangles per peg. So that's a lot of memory and structure. When a peg moves across the board, we have to change its coordinates. That costs a lot of performance and time. It was the first way we did. The following output shows the garbage collection by translating all vertices for each move and of course refilling the buffer.

(1) Manual translation

01-26 16:45:29.816: D/dalvikvm(4018): GC\_CONCURRENT freed 878K, 54% free 3213K/6855K, external 2028K/2108K, paused 4ms+6ms  
01-26 16:45:30.613: D/dalvikvm(4018): GC\_CONCURRENT freed 909K, 54% free 3220K/6855K, external 2031K/2108K, paused 3ms+5ms  
01-26 16:45:31.398: D/dalvikvm(4018): GC\_CONCURRENT freed 916K, 54% free 3179K/6855K, external 2025K/2108K, paused 4ms+7ms  
01-26 16:45:32.203: D/dalvikvm(4018): GC\_CONCURRENT freed 888K, 54% free 3187K/6855K, external 2022K/2108K, paused 4ms+6ms  
01-26 16:45:32.988: D/dalvikvm(4018): GC\_CONCURRENT freed 888K, 54% free 3216K/6855K, external 2029K/2108K, paused 7ms+6ms  
01-26 16:45:33.777: D/dalvikvm(4018): GC\_CONCURRENT freed 922K, 54% free 3189K/6855K, external 2028K/2108K, paused 4ms+7ms  
01-26 16:45:34.550: D/dalvikvm(4018): GC\_CONCURRENT freed 914K, 54% free 3163K/6855K, external 2021K/2108K, paused 4ms+9ms  
01-26 16:45:35.335: D/dalvikvm(4018): GC\_CONCURRENT freed 873K, 54% free 3174K/6855K, external 2020K/2108K, paused 3ms+9ms  
01-26 16:45:36.124: D/dalvikvm(4018): GC\_CONCURRENT freed 878K, 54% free 3192K/6855K, external 2025K/2108K, paused 4ms+7ms  
01-26 16:45:36.910: D/dalvikvm(4018): GC\_CONCURRENT freed 907K, 54% free 3173K/6855K, external 2024K/2108K, paused 5ms+7ms

That means that in 7.094 seconds, we pause the game for 111 ms, which is about 15.6 ms/s. Also we have to clear about 8973K (1264.9K/s). So you can see, that there is a lot of waste. Of course we don't know, whether this is waste made by the translation of the coordinates. OpenGL es has functions for translating and also rotating, that we need anyway, because of the rotating dice. So we changed the structure. Now the game object has the already mentioned position and the offset is added intern during the rendering. So all pegs have the same vertices and we just change three float values for x, y and z-position of an object. OpenGL adds that offset to the mesh. In fact we change the intern matrix of OpenGL. Now we don't want to change it, because the rest of the world has to stay untouched. For this action OpenGl can push() the matrix, meaning that a copy is created and put on top of OpenGl's matrix stack. With pop() put the matrix back and merge them. All changes after the push won't effect the previous settings. Now here is another output of the garbage collection.

(2) Push and pop instead of manual translation

01-26 16:53:26.105: D/dalvikvm(4073): GC\_CONCURRENT freed 557K, 60% free 2699K/6663K, external 1898K/2108K, paused 9ms+2ms  
01-26 16:53:43.050: D/dalvikvm(4073): GC\_CONCURRENT freed 557K, 60% free 2699K/6663K, external 1898K/2108K, paused 8ms+2ms  
01-26 16:53:59.976: D/dalvikvm(4073): GC\_CONCURRENT freed 557K, 60% free 2699K/6663K, external 1898K/2108K, paused 8ms+2ms  
01-26 16:54:16.867: D/dalvikvm(4073): GC\_CONCURRENT freed 556K, 60% free 2699K/6663K, external 1898K/2108K, paused 2ms+2ms  
01-26 16:54:33.812: D/dalvikvm(4073): GC\_CONCURRENT freed 558K, 60% free 2698K/6663K, external 1898K/2108K, paused 8ms+3ms  
01-26 16:54:50.761: D/dalvikvm(4073): GC\_CONCURRENT freed 556K, 60% free 2698K/6663K, external 1898K/2108K, paused 8ms+1ms  
01-26 16:55:07.679: D/dalvikvm(4073): GC\_CONCURRENT freed 556K, 60% free 2699K/6663K, external 1898K/2108K, paused 8ms+2ms  
01-26 16:55:24.609: D/dalvikvm(4073): GC\_CONCURRENT freed 557K, 60% free 2699K/6663K, external 1898K/2108K, paused 8ms+3ms  
01-26 16:55:41.597: D/dalvikvm(4073): GC\_CONCURRENT freed 557K, 60% free 2698K/6663K, external 1898K/2108K, paused 8ms+1ms  
01-26 16:55:58.515: D/dalvikvm(4073): GC\_CONCURRENT freed 555K, 60% free 2699K/6663K, external 1898K/2108K, paused 8ms+1ms

As you can see, the first ten cleanings are done in more than two and a half minutes. The result of 152.41 s is pausing for 94 ms, which is 0.6 ms/s and cleaning 5566K (36.5K/s). Now we know that our improvements made a big change, because pausing was decreased 26 times and the data to clear is about 35 times less than before.

Now all classic pegs have the same coordinates, but a different position. Because of that, we can simple make one static buffer for all pegs. We can do the same with colors and textures, as well as the colors for selection. That saves lots of memory, too. And that's how it comes, that our classic peg is in charge of all geometry-related issues. The rendering itself is still done in the simple geometric object. And talking about coordinated, colors and other resources, we have to mention that for the hole board and all pegs as well as the dice, we don't have any resources at all. Everything is created on game start and saved static final. That means, we can easily change the shape of a peg or adjust the team colors.

Another feature of our game is, that the textures are exchanged during the game. Because the user can change the view of the camera and rotate it around the board, you might get closer to some textures. Also the user can enable multi-touch and zoom into the world, getting closer to the origin and to the board's surface. So we made up some texture control system and when the view is changed, we check the distance to the textures. If we got closer, we replace the texture-id with one, that is bound to a bigger bitmap. So we always get round shapes, but never need too much performance, because if we zoom out, we just replace the texture with a smaller one. By the way, texture's side has to be a power of two. Our board textures are created in five different sizes on start.

## Bluetooth Communication

### Classes

Multiplayer version of the game uses Bluetooth for the communication between devices. Following graphs show the class diagrams for the package de.tum.multiplayer, which is responsible for the multiplayer version of the game including the Bluetooth communicatioon.

### MultiplayerActivity

After user selects the Multiplayer version of the game from the WelcomeActivity screen, MultiplayerActivity is created. Upon creation of the activity, board and dice object are drawn to allow the upcoming dialog activities use it as background view for better user experience. Other related classes and threads communicate with MultiplayerActivity via onActivityResult() and the Handler object, respectively. Under the titles named “Communication with the Main Activity” and “Actions According to the Activity Results”, you can find the details of this communication written on the code snippet.

### ModeSelectionActivity

Right after the MultiplayerActivity is created, another activity named ModeSelectionActivity comes as a dialog screen. This activity allows the user to select the device mode, and make the current device discoverable for other devices to allow pairing if they haven’t been paired before. One device might be server or client.

### ClientNumberPicker

If user selected to make his device as server device at the ModeSelectionActivity, next activity will be ClientNumberPicker activity. This allows the server device user to select how many clients will connect it to play the multiplayer mode. Game application supports 7 more devices to connect to server device. However, only 3 of them can play the game while the rest can follow/watch the game on their devices’ screen.

|  |
| --- |
| **Note:** During our tests with different devices and different Android OS versions, we have seen that only the devices whose Android version is 2.3.5 or above can accommodate up to 7 devices as client, while earlier versions of Android only able to accept 2 client devices. Therefore, it’s suggested to make the device whose Android version is 2.3.5 or above be as client if the users are planning to play with more than 3 devices total. |

### DeviceListActivity

If user selected to make his device as client device at the ModeSelectionActivity, next activity will be DeviceListActivity displaying the already paired devices and allows scan more devices which are discoverable. Therefore, user won’t need to switch from game to his device’s Bluetooth settings to pair his device with others. Upon selection of device name form the list. MAC address will be passed to already initialized BluetoothMPService, and client’s connection attempt to server device will start.

### BluetoothMPService

This is the class where all the connection is handled via Threads. You can find the detail explanation at the following section with its sequence diagrams.

### DataTransfer

This class’s objcets represents the data sending to other devices. It has fields to accommodate game’s state.

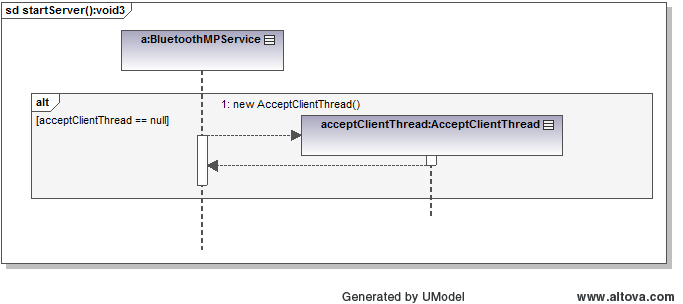
### TeamMatching

This is the activity responsible for team properties selection after all the devices are connected to the server device.

|  |
| --- |
| C:\Users\Baris\Desktop\de.tum.multiplayer.jpg |
| Class Diagram for the package de.tum.multiplayer |

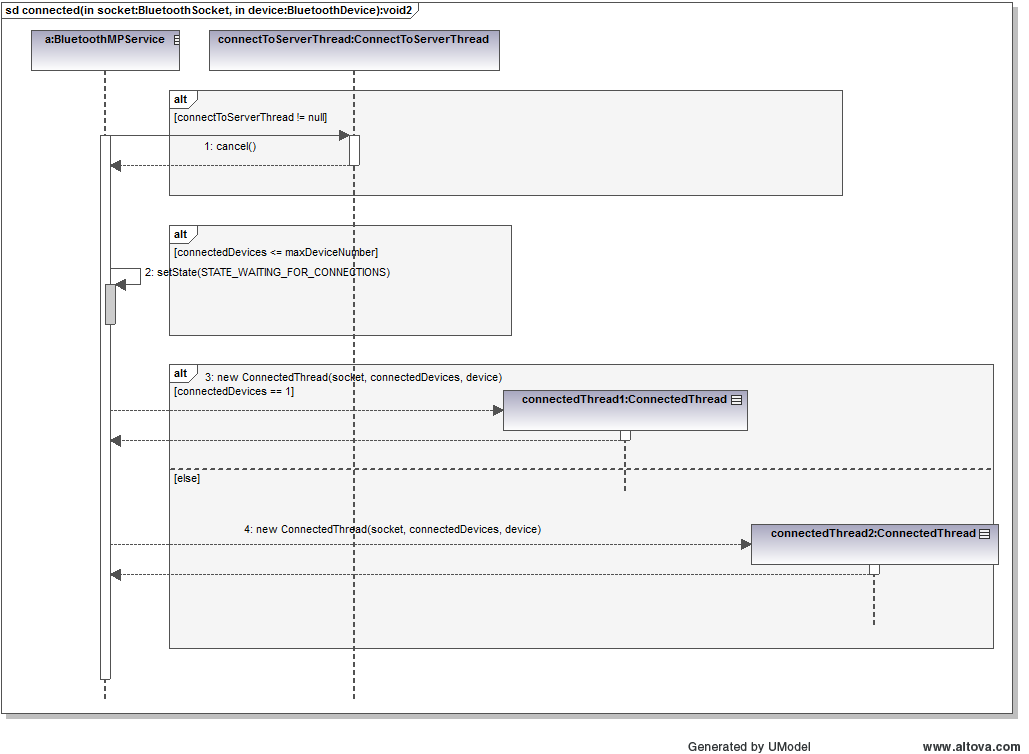
## Server and Client Side Bluetooth Communication

### Server Side

After user, who is using the server device, decides how many clients devices will be available for the game and decides the properties of teams, startServer() method is called from the BluetoothMPService class. This method creates a thread (AcceptClientThread) for listening upcoming client connections to the server device. At this moment, server device user is informed with a progress dialog box for each connected devices.

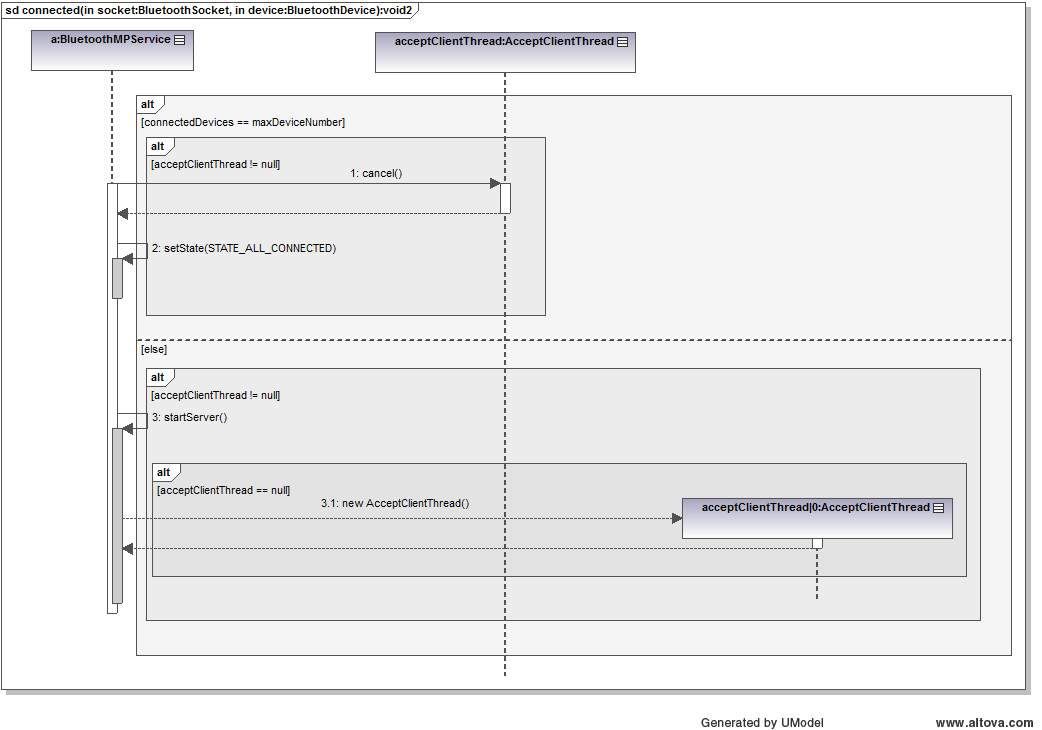
Creating a thread for listening upcoming client connections to the server device.

After a client requests a connection to the server device, another thread (ConnectedThread) is created for dealing with the communication between devices. If the maximum number for the connected devices is not reached, state of the service is set to “STATE\_WAITING\_FOR\_CONNECTIONS”, and the thread for accepting new connections (AcceptClientThread) is recreated. Following sequence diagram is the simplified version for server accepting two client devices.



Accepting client a connection and waiting for more devices.

Accepting client connection thread runs until all the devices are connected to the server device (When connectedDevices is equal to maxDeviceNumber), and after each connected device it is reinitialized to make the BluetoothSocket free. For each of the 7 clients there is a unique UUID, which is incremented after each connected device. When all the client devices are conencted, the state is set to “STATE\_ALL\_CONNECTED”, and letting the MultiplayerActivity know with handling a Message, and another activity (TeamMatching) starts to let the server device user assign the team properties. Afterwards game begins. Following sequience diagram shows those method calls in the BluetoothMPService class.



Checking if all devices are connected or not.

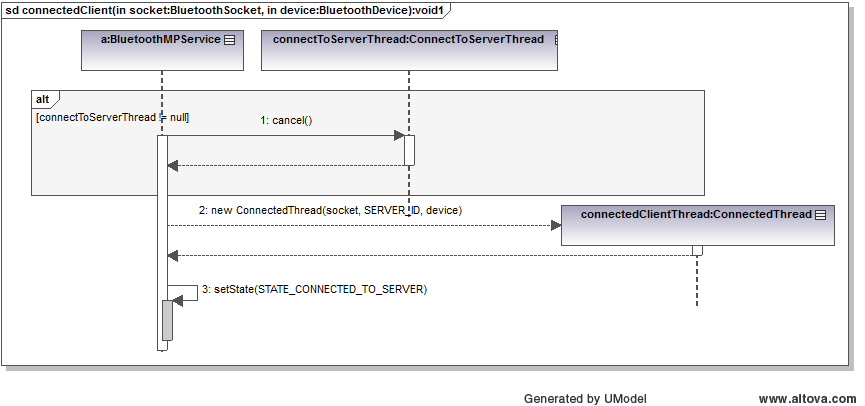
Following table shows the states while communicating with the main activity (MultiplayerActivity). Messages are sent via Handler initiated in the MultiplayerActivity, and handled to the BluetoothMPService when it is initialized.

|  |  |
| --- | --- |
| **STATE\_NONE** | When BluetoothMPService class is initiated |
| **STATE\_LISTEN** | When server is listening for incoming client connections |
| **STATE\_WAITING\_FOR\_CONNECTIONS** | When one or more clients are already connected to server, but waiting for more devices |
| **STATE\_ALL\_CONNECTED** | When all the devices are connected to server, and ready to start the game. |

### Client Side

When the user of the device selects the client mode, the next activity will be the screen where the paired devices list and unpaired devices list appear. User can also select to scan for more devices by clicking the button named “scan for devices”. If both devices are not paired before, the server machine user can select the option “Make This Device Discoverable” before switching to server mode. Hence server machine will be discoverable for 300 seconds, afterwards client device user can pick the server name from the list to be paired and complete the device selection activity (DeviceListActivity).

Following sequence graph reflects the activities done after the user decides his machine to be client. Selected server device’s MAC address is passed to the thread which is establishing connection by probing the socket with corresponding UUID. If one of the UUID is failed due to already reserved for another device on the server side, clients continue with probing the next UUID on the Socket. After the connection is established, another thread is created (connectedClientThread:ConnectedThread) to start the data transfer between the server and itself.



Client device connects to server device

Following table shows the states to communicating with the main activity which is MultiplayerActivity. Messages are sent via Handler created in the MultiplayerActivity, and handled to the BluetoothMPService when it is initialized.

|  |  |
| --- | --- |
| **STATE\_NONE** | When BluetoothMPService class is initiated |
| **STATE\_CONNECTING\_TO\_SERVER** | When probing the UUID on the opened Socket |
| **STATE\_CONNECTED\_TO\_SERVER** | When Socket is accepted and connected to the Server device |

## Communication with the Main Activity

Following table describes the messages arriving from BluetoothMPService, and the actions according to message type.

|  |
| --- |
| *1) MESSAGE\_STATE\_CHANGE* |
| Connection state is changed |
| *1.1) STATE\_ALL\_CONNECTED* |
| All the devices are connected to the server, so start the game, and set the title to “All connected”  *1.2) STATE\_WAITING\_FOR\_CONNECTIONS* |
| Toast the connected device number (Server side) |
| *1.3 and 1.4) STATE\_CONNECTED\_TO\_SERVER and STATE\_CONNECTING\_TO\_SERVER* |
| Set the title on the client side to "Connecting to server..." |
| *1.5 and 1.6) STATE\_LISTEN and STATE\_NONE* |
| Nothings happen (for debugging purposes and extension for new versions) |
| *2) MESSAGE\_WRITE* |
| The message that this device has been sent to other(s) (for debugging purposes) |
| *3) MESSAGE\_READ* |
| Message has arrived to this device from other(s). Convert it to do the corresponding object depending on if this is client or server device |
| *4) MESSAGE\_DEVICE\_NAME* |
| **Server Side:** A client connected to the server. Change the value of the progress bar on the server side according to total number of connected devices.  **Client Side:** Toast the name of the server device |
| *5) MESSAGE\_TOAST* |
| Game state has been changed, and letting the player know about the command that he has to do. |
| *6) MESSAGE\_TOAST\_WARNING* |
| Warn the user about the connection lost or connection failed situation, and bringing options menu to the screen to let the user continue to the game as client or server again. |
| *7) MESSAGE\_TITLE* |
| Change the text on the title bar with message arrived from BluetoohMPService |

## Actions According to the Activity Results

Following table describes the actions after the results come back from the Activities started in the MultiplayerActivity.

|  |
| --- |
| *1) REQUEST\_CONNECT\_SERVER* |
| Result comes from ModeSelectionActivity. MAC address of the server selected device will be used to probe the connection in the BluetoothMPServer’s connectServer() method. |
| *2) REQUEST\_ENABLE\_BT* |
| If Bluetooth is not enabled on the device, application will enable the Bluetooth. If result is not positive, then user will be prompt to continue with single player mode. |
| *3.1) REQUEST\_MODE\_TYPE* : *RESULT\_CLIENT\_MODE* |
| User selected to be client on his device at the ModeSelectionActivity. Now, another activity named DeviceListActivity will let the user to select the name of the server device to connect. |
| *3.2) REQUEST\_MODE\_TYPE* : *RESULT\_SERVER\_MODE* **if** (clientNumberPickerIntent == **null**) |
| User selected to be server on his device at the ModeSelectionActivity. Now, another activity named ModeSelectionActivity will let the user to select the number of the client devices to play with. |
| *3.3) REQUEST\_MODE\_TYPE* : *RESULT\_SERVER\_MODE* **if** (clientNumberPickerIntent != **null**) |
| Server device’s user selected how many clients will be available. Now, this information will be set in the BluetoothMPService as well, and it’s server process will be started. Then a progress bar will be created to show the states of the connections to the server device. |
| *4 and 5) RESULT\_CANCELED* and *RESULT\_GOBACK* |
| Result when an Activity is ended with pressing back button before it does its job. So, the MultiplayerActivity will be ended as well. |

# Detailed Test Plan

## Devices that we tested on

* HTC Desire HD (Android 2.3.3)
* HTC Desire HD (Android 2.3.5)
* Samsung Galaxy S2 (Android 2.3.4)
* Samsung Galaxy S (Android 2.3.3)
* Nexus One (Android 2.3.6)
* Nexus One (Android 2.3.7)

## Testing different screen resolution and size

We tested our game application different activity screens on different screen sizes to see how it fits to the screen. We mainly used the different screen size options given in the Android’s graphical layout development tool in Eclipse IDE.

## Testing Multiplayer Mode

We tested the multiplayer mode by assigning different devices with different Android platform as a server device, and connection to other devices to it in different combinations. We interrupted the connection of one connected client device from the server to test the game flow.

# Project Conclusion

At the end of successful development period, we have had a nice board game having multiplayer support. We have seen many challenges that make us to find the solutions efficiently. This project has not only resulted a nice board game, but also a good development structure to initiate other projects.