Software Design Document for: Siege Towers

Feel Good Inc.

Version 8.0

March 3rd, 2011

Table of Contents

Revision History

Name	Date	Reason	Version
Steven Edouard	2/19/10	Initial Creation	1.0
Jake Cohen	2/25/10	Added template sections	2.0
Michael and Becca	2/28/10	Added UI/graphics section	3.0
Alex and Andrew	3/1/10	Added Networking	4.0
Jake and Christian	3/2/10	Added Build phase	5.0
Steven and Brett	3/2/10	Added Game play phase	6.0
Entire team	3/3/10	Added UMLs and sequence diagrams	7.0
Jake Cohen	3/4/10	Formatting	8.0

1. Introduction

The purpose of this document is to convey the design philosophy and architecture of the Siege Towers application. This document defines how the developers intend to implement the application to function according to the software requirements previously submitted. The application is intended to provide entertainment among users through competitive game play on the Android platform.

1.1 Goals and Objectives

The goal of the Siege Towers development team is to design and develop a turn-based, multi-player game for the mobile android platform. The game will be two-dimensional and will incorporate simplistic graphics and physics engines. A remote server will be used to store game and player information and to facilitate game communication among clients. The game will be limited to two players.

1.2 Statement of Scope

In Siege Towers, players build a tower with blocks of various colors and sizes in a limited amount of time. The goal is to destroy the opponent's tower with user-selected weapons. It is a mobile Android OS-based, turn-based, multi-player game. The game will be invitation-based but the web server could be used to search a database for available opponents. The following are features listed in the SRS that we are implementing:

1.2.1 AndEngine Graphics Library

This open source GL will be the most essential part of our game. It will handle graphics rendering, sprites, physics, and the UI.

1.2.2 Handling Physics

Siege Towers will handle physics using AndEngine. Physics will be used during the Build and Game phase.

1.2.3 Multi-player Capability

This feature allows the user to play other users on different android devices.

1.2.4 Multiple Games at Once

This feature allows the user to play multiple games at once.

1.2.5 Collision Detection

Collision detection will be an integral part of the game. This functionality will be used to determine when a Weapon hits a TowerBlock. Collision detection is performed by AndEngine.

1.2.6 Server Client Hosting

Communication between clients will be intermediated by a remote Linux server. The server will contain a database that stores all game and player information. The server will communicate with client applications through an http interface.

1.2.7 Push Notifications

Cloud to Device Messaging (C2DM) will be used in the application via the supplied Google API. Google, who provided a simple API for implementation, introduced this new feature in Android 2.2. This methodology will prove very efficient, especially in terms of power consumption, as the Siege Towers client will not need to have processes running in the background of the Android OS in order to make sure that the user stays up to date. This avoids using a polling strategy, making the client application much more efficient in terms of power consumption.

1.3 Software Context

This application is being developed to release onto the Android Application Marketplace. Siege Towers will generate revenue through either advertisement placed in a free application or by charging a small fee to download and install the application. After a successful release into the Android market a separate iPhone application can be developed to allow cross platform multiplayer Android iOS games to be played.

1.4 References

"**AndEngine**." *AndEngine - Free Android 2D OpenGL Game Engine*. Web. 04 Mar. 2011. <http://www.andengine.org/>.

"**Android Cloud to Device Messaging Framework - Google Projects for Android**." *Google Code*. Web. 04 Mar. 2011. <http://code.google.com/android/c2dm/index.html>.

"**Platform Versions**." *Android Developers*. Web. 04 Mar. 2011. <http://developer.android.com/resources/dashboard/platform-versions.html>.

"**Rock Paper Scissors Game Basics**." *World RPS Society*. Web. 04 Mar. 2011.

<http://www.worldrps.com/gbasics.html>.

"**Rovio - Angry Birds**." *Rovio - Home*. Web. 04 Mar. 2011.
<http://www.rovio.com/index.php?page=angry-birds>.

"**WordFeud**." *Wordfeud - Multiplayer Word Game for Android*. Web. 04 Mar. 2011.
<http://wordfeud.com/>.

"**Words With Friends**." *Zynga With Friends*. Web. 04 Mar. 2011. <http://newtoyinc.com/wp/>.

1.5 Major Constraints

Due to the unique nature of this project, an academic implementation of a full-blown commercial game project, this software will have several uncontrollable constraints.

Project Size: The staffing for this project for is fixed by enrollment in the course. Therefore more developers cannot be added if the scope of this software expands. This will limit the number of features that can easily be implemented. To mitigate the project size constraint clearly defined groups with objectives and goals have been created for the sub-groups.

Project Timeline: The deadline for this software is absolutely fixed by the end of the semester. This will result in constraints on the features that can feasibly be implemented. To overcome this constraint a comprehensive schedule of implementation and deadlines will be created to ensure that all necessary features will have sufficient time allocated to their development.

2 Data Design

A description of all data structures including internal, global, and temporary data structures.

2.1 Internal Software Data Structure

2.1.1 Game Class

The Game class stores information about a single game being player. The game class will contain all necessary information to represent a game. The constructor for this class takes no variables. The variables that the class holds are:
int myNumber //user's phone number
int opponentNumber //opponent's phone number
String myUserName //user's username

int gameID //unique ID given to each game
Enum.GameResult mResult //the end result of the game
int myHealth //percentage of user's tower still intact
int opponenetHealth //percentage of opponent's tower still intact
int myCriticalBlows //number of user's hits that resulted in more than x% damage, used in statistics shown at the end
int oppCriticalBlows //number of opponent's hits that resulted in more than x% damage, used in the statistics shown at the end
long startTime //what date and time the game was started
boolean isDirty //is true if this object has been modified
int mWeapon1 //number of Weapon1 user has
int mWeapon2 //number of Weapon2 user has
int mWeapon3 //number of Weapon3 user has

2.1.2 TowerBlock Class

The TowerBlock class holds the information about each block that is used, including its placement on the screen.
float pX //x-coordinate of block in tower
float pY //y-coordinate of block in tower
TiledTextureRegion pTiledTextureRegion //texture region for block placement
Body pBody //the body that the physics is being applied to
PhysicsWorld pWorld //the physics environment
Defense pDefenseType //color of block
int pHitPoints //how many points until a block is destroyed

2.1.3 WeaponsOverlay Class

The WeaponsOverlay class contains the number of weapons in the game as well as the type of each weapon. The constructor for this class takes no variables.
The variables that the class holds are:
int mWeaponCount //number of total weapons for the game
WeaponType mWeaponType //the type of weapon

2.1.4 NetworkManager Class

The NetworkManager class will handle talking to the network and querying the database for SiegeTowers.
The constructor for this class takes no variables.

2.1.5 BlockGeneratingSprite Class

The BlockGeneratingSprite class is responsible for displaying the blocks on the screen.

float pX //x-coordinate of block in tower

float pY //y-coordinate of block in tower
TiledTextureRegion pTiledTextureRegion //texture region for block placement
Body pBody //the body that the physics is being applied to
PhysicsWorld pWorld //the physics environment
Shape pShape //an interface, this is a superclass of Sprite

2.1.6 GamePlayActivity Class

The GamePlayActivity class contains all the current information about the gameplay, including the tower setups, weapon counts, and many of the sprites on the screen.
The constructor for this class takes no variables. The variables that the class holds are:
ArrayList<TowerBlock> mMyTowerBlocks // stores the towerBlocks in user's tower
ArrayList<TowerBlock> mOpponentTowerBlocks //store the towerBlocks in opponent's tower
ArrayList<Weapon> mMyWeapons // stores user's weapon selections
ArrayList<Weapon> mOpponentWeapons //stores opponent's weapon selections
Game mThisGame // stores the game object
ChangeableText mMyText //stores user's health
ChangeableText mYourText //stores opponent's health
ZoomCamera mCamera //the camera used to pan screen
TimerHandler mGameUpdater //automatically refresh
Scene mScene //the scene that sprites are mounted to
The following sprites are all stored in this class:
AnimatedSprite mWeaponSelector
AnimatedSprite mMyHealthBar
AnimatedSprite mOpponentHealthBar
AnimatedSprite mMySlingShot
AnimatedSprite mOpponentSlingShot
AnimatedSprite mTurnArrow

2.1.7 Weapon Class

The Weapon class gives a weapon type to each weapon created. The constructor for this class takes no variables. The variables that the class holds are:
Enum.WeaponType mWeaponType //holds the type of weapon

2.1.8 C2DMHandler class

The C2DMHandler class holds a log tag and is responsible for Push notifications. The constructor takes no variables.
Static final String TAG // Log Tag

2.2 Global Data Structure

Data structures that are available to major portions of the architecture are described.

2.2.1 Description of GameManager Class

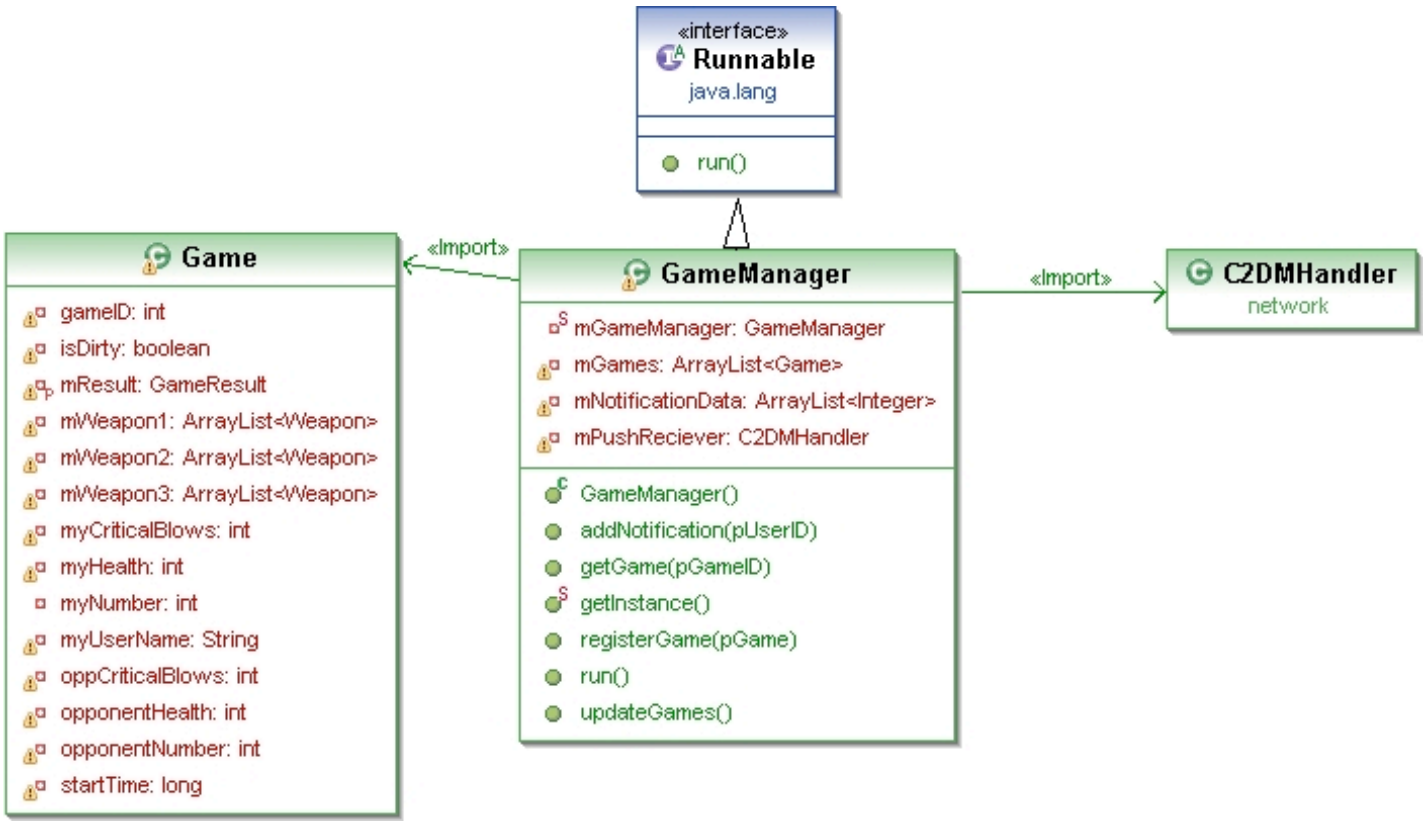
The GameManager class is a background service class. It runs in a separate thread from the rest of the Siege Towers client code. The game manager run a producer-consumer algorithm to check the status of all games, and will transmit or receive information from the network as necessary.

2.2.1.1 Processing Narrative (PSPEC) for GameManager Class

The GameManager class will be responsible for creating new games and keeping track of existing games. This class will act as the central thread of the application around which all other activities are organized. Game updates sent to and received from the network will be handled in this thread. These updates could result from an opponent taking a turn, a game starting or ending or any network failures. As the user interacts with the various interface activities all information generated by the user will be sent to the GameManager thread for processing and storage.

2.2.1.2 GameManager Class Interface Description

Since the GameManager is a continuously running consumer thread it will have limited interfaces. The producers will be the NetworkManager class and Activities classes (such as TowerBuildActivity.) The interface to this class will be through its synchronized ArrayList of Game objects. As game state changes, from either a network update or a user interaction, the GameManager class will update the Games ArrayList.



2.2.1.3 GameManager Class Processing Detail

A detailed algorithmic description for each component is presented below.

2.2.3.1.1 Design Class Hierarchy for GameManager Class

The GameManager class will implement the Java Interface runnable. This will allow games to be updated concurrently by either the server or the user. The GameManager is a singleton accessible to every component of SiegeTowers.

2.2.3.1.2 Restrictions/Limitations of GameManager Class

There should only be a single instance of the GameManager class at any point in the application. If several GameManager classes were running the game state could be altered in unpredictable ways.

2.2.3.1.3 Performance Issues of GameManager Class

There are no expected performance issues with this class. It is a single running thread and all available android handsets will be capable of running it.

2.2.3.1.4 Design Constraints for GameManager Class

The GameManager class is intended act as an architecture layer between the network and the user interface activities. Therefore the GameManager will never interact directly will the user or will have no visual components.

2.2.3.1.5 Processing Detail for Each Operation of GameManager Class

CreateGame() : Will add the specified game to the Games ArrayList and will update the network that a new game is being played.

getGame() : Searches the Games ArrayList and returns the game with the gameID specified if found.

2.2.2 Description for Enumerations Class

The enumerations class is a storage class for enumerations used to hold all of the enumerations that our application requires. The enumeration class has no methods and its only functionality is as a storage class.

2.2.2.1 Processing Narrative (PSPEC) for Enumerations Class

The Enumerations class is responsible for providing all enumeration needs for each component in Siege Towers. This includes enumerations for:

- Weapons
- Defense
- Block Shape
- Game State
- Game Results

2.2.2.2 Enumerations Class Interface Description

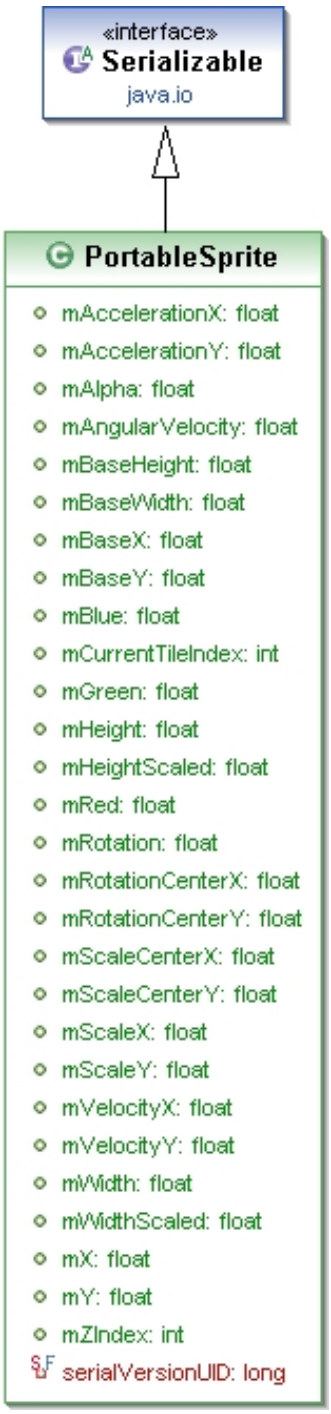
2.2.2.3 Enumerations Class Processing Detail

- **Defense** - The defense level of each tower block. Each defense level reacts differently with each weapon type.
- **Game Result** - The result of the game after completion.
- **Game State:**

        - INVITE - Invitation has been sent from this user and is waiting for response.
        - IN_BUILD - This user has not completed or in progress of competing their tower.
        - BEGIN_BUILD - This user not begun construction on their tower.
        - READY - This user has created their tower and is waiting for the opponent to complete their tower.
        - COMPLETED - The game has completed.
        - YOUR_TURN - It is currently this player's turn to make a move.
        - WAITING_FOR_OPPONENT - It is currently the opponent's turn to make a move.
    - **WeaponType**: An enumeration for the 3 separate weapons used in Siege Towers
        - WEAPON_1 - Strong against WEAPON_2, weak against WEAPON_3
        - WEAPON_2 - Strong against WEAPON_3, weak against WEAPON_1
        - WEAPON_3 - Strong against WEAPON_1, weak against WEAPON_2
    - **Shape**: The physical shape of the Tower Block sprite
        - LARGE_RECTANGLE - This block will be a large rectangle
        - SMALL_RECTANGLE - This block will be a small rectangle
        - SQUARE- This block will be a large square

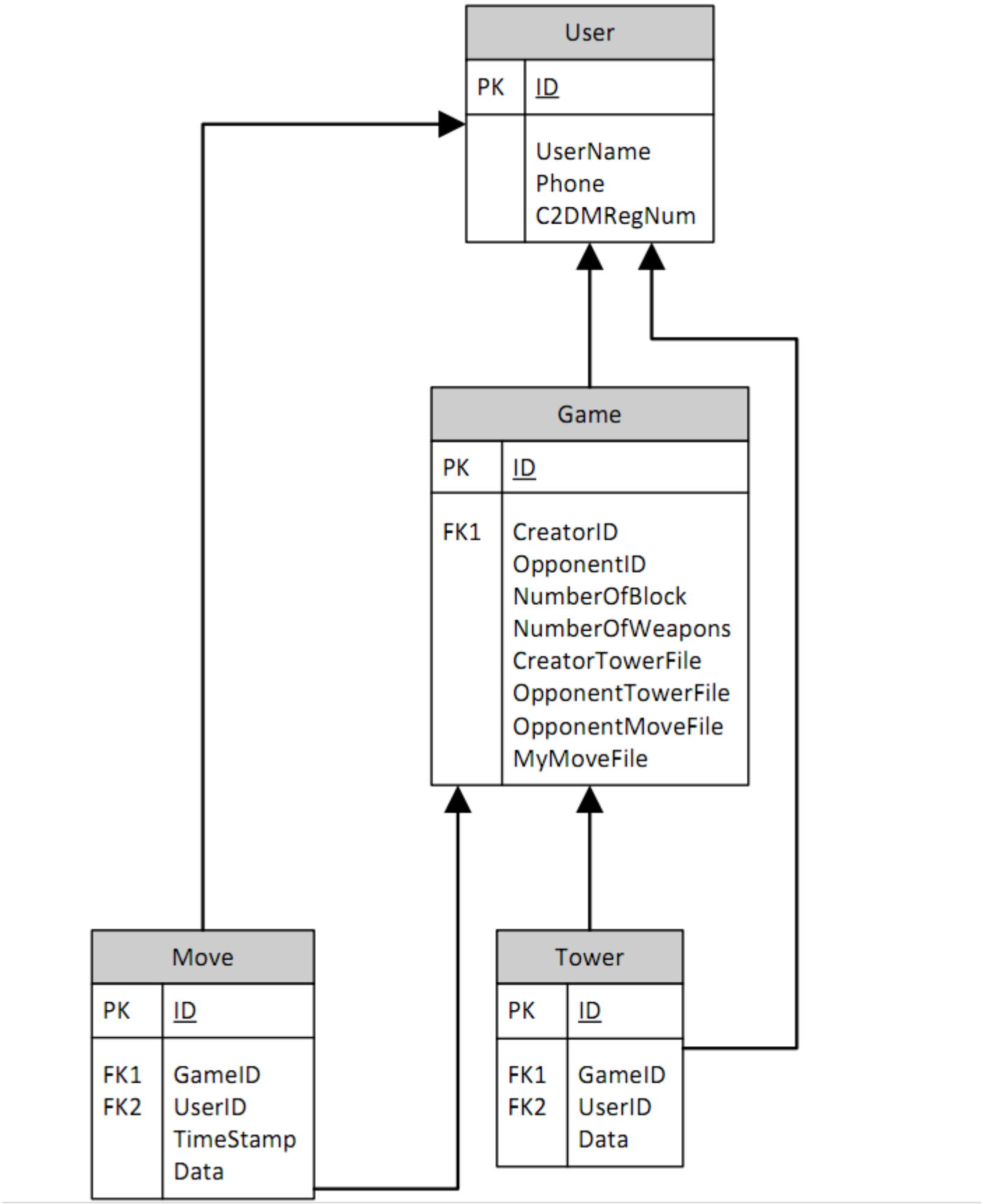
2.3 Temporary Data Structure

Class PortableSprite is a structure, which holds attributes needed to flatten or reconstruct a sprite. It contains fields for all publicly accessible fields in Class AnimatedSprite and implements the Java Serializable interface. This class is used to create a ".ser" file to be saved for later use or sent via network interfaces.



2.4 Database Description

The SiegeTowers game server MySQL database stores user data, game data, tower data and the most recent move by each opponent. The database schema is purposefully simple and restrained; rather than storing all parameters of objects, Serialized objects will be stored in the case of the Towers and Moves. Class methods are


User
PK
ID
UserName
Phone
C2DMRegNum
Game
PK
ID
FK1
CreatorID
OpponentID
NumberOfBlock
NumberOfWeapons
CreatorTowerFile
OpponentTowerFile
OpponentMoveFile
MyMoveFile
Move
PK
ID
FK1
FK2
GameID
UserID
TimeStamp
Data
Tower
PK
ID
FK1
FK2
GameID
UserID
Data

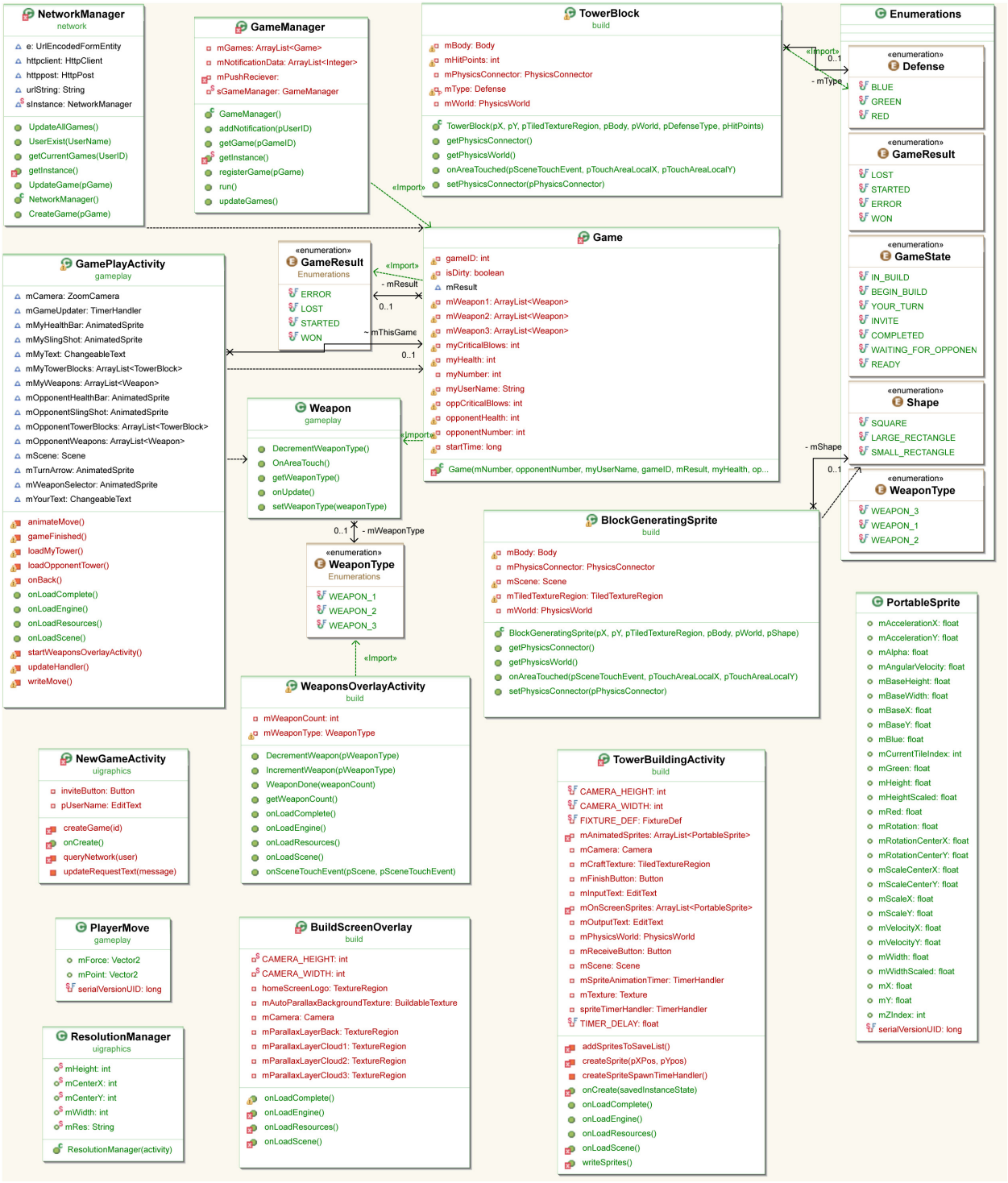
3.0 Architectural and Component-Level Design

Our Android Project Siege Towers will be built with the Google Android Development Kit for Java. It will be implemented with Java and XML. Siege Towers will be implemented using an open source graphics library called AndEngine. This will handle sprite generation, physics environment, and game graphics. The base controlling classes will be SiegeTowers and GameManager. The class SiegeTowers will be the upper level class called at the start of the program. The GameManager class will manage each game the user is in. From SiegeTowers, any activity that is started will handle it's own data and user interactions in its corresponding class.

3.0.1 System Structure

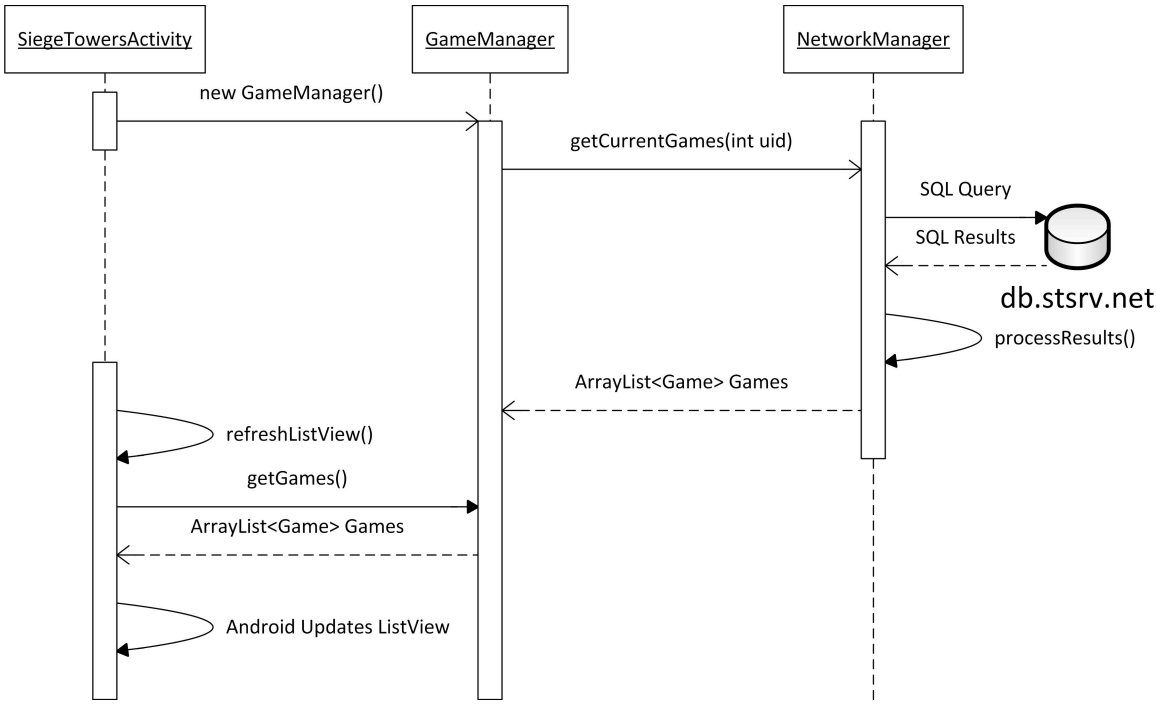
The game will be implemented using the 2-Dimensional game engine: AndEngine. From this game engine in combination, with the Android SDK, the game will progress through the Game manager and Network manager.

3.0.2 Architecture Diagram

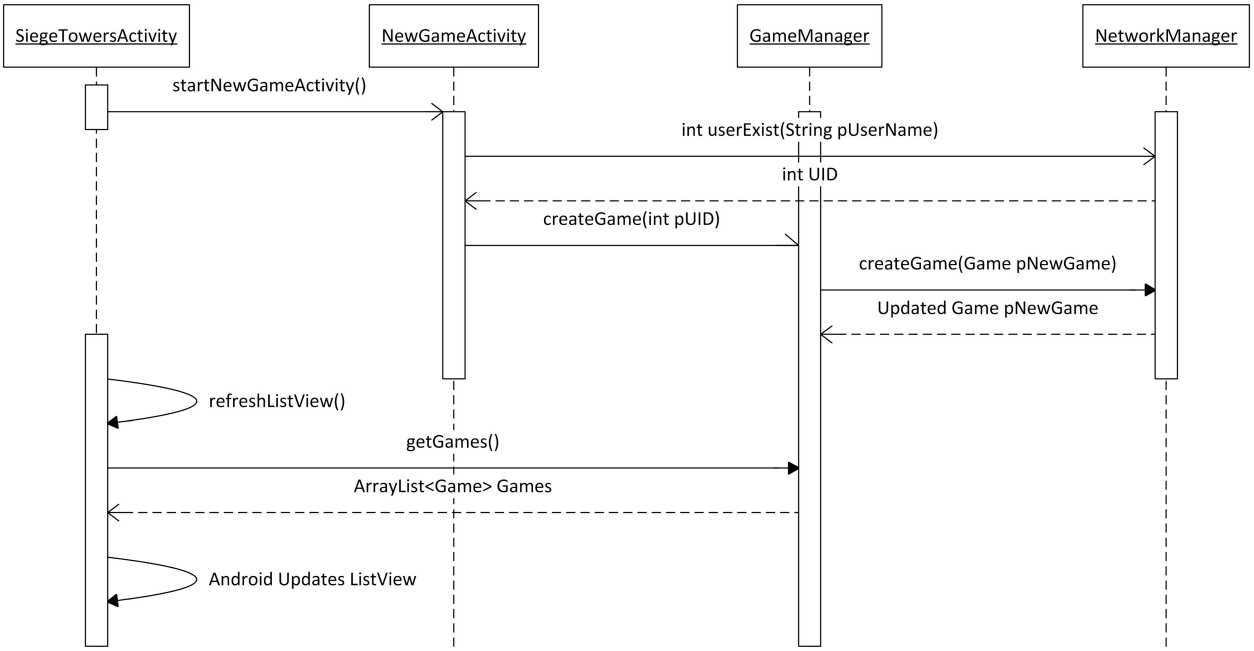


3.0.3 Sequence Diagrams

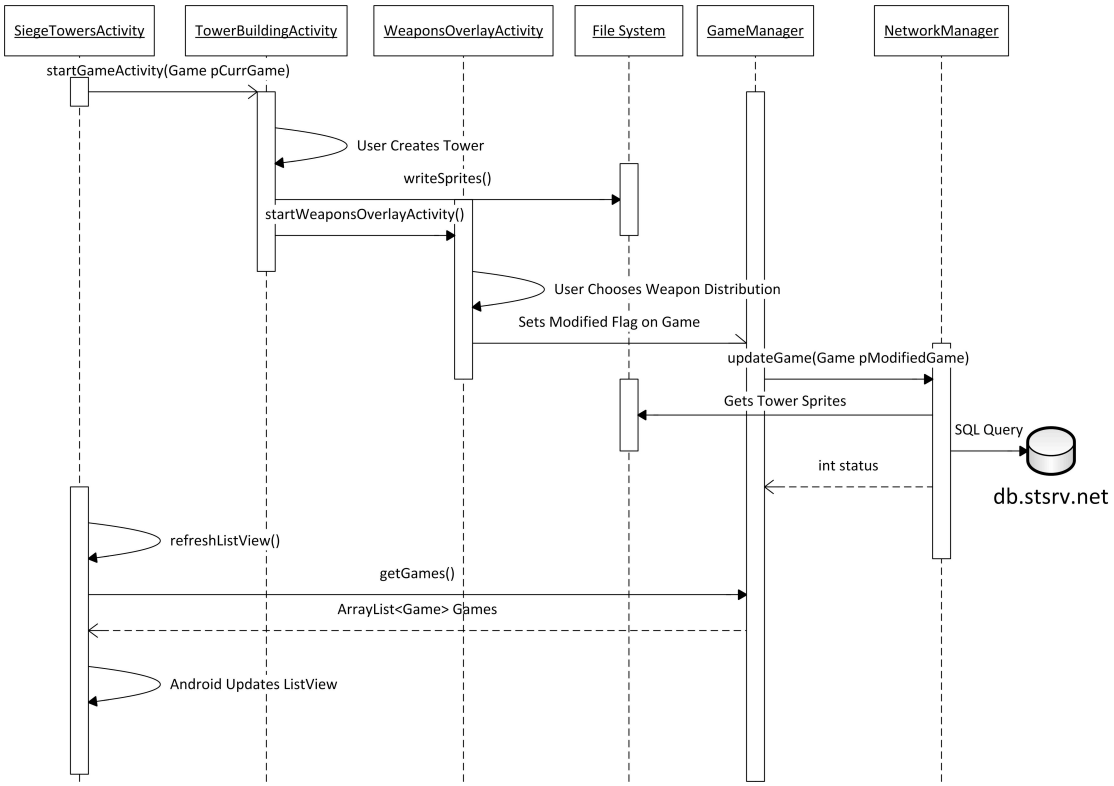
3.0.3.1 Application Launch Sequence



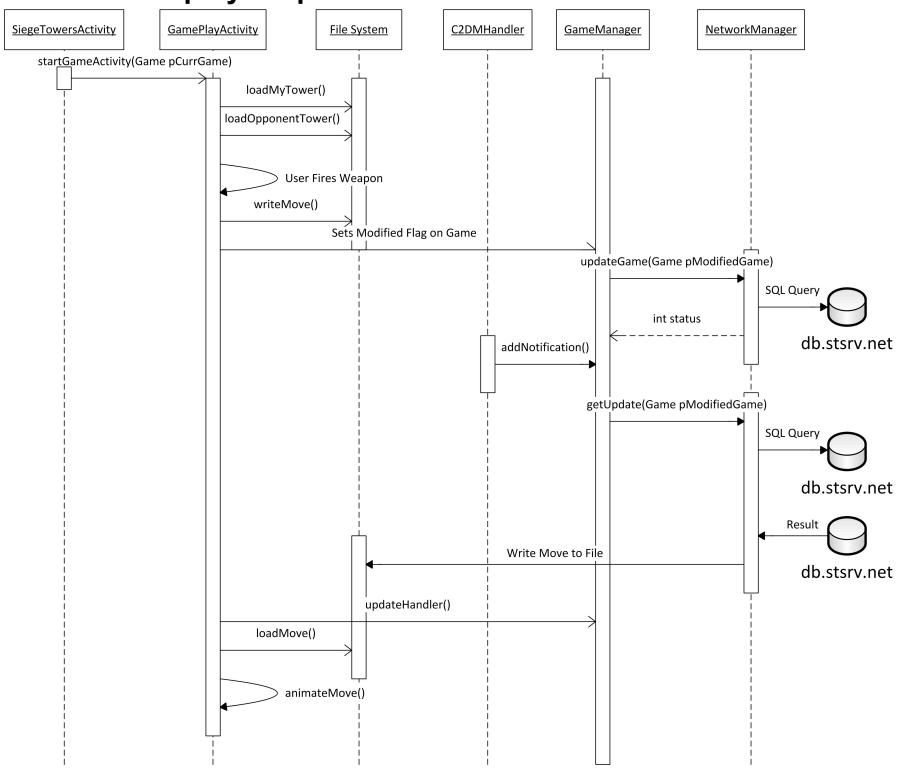
3.0.3.2 New Game Sequence



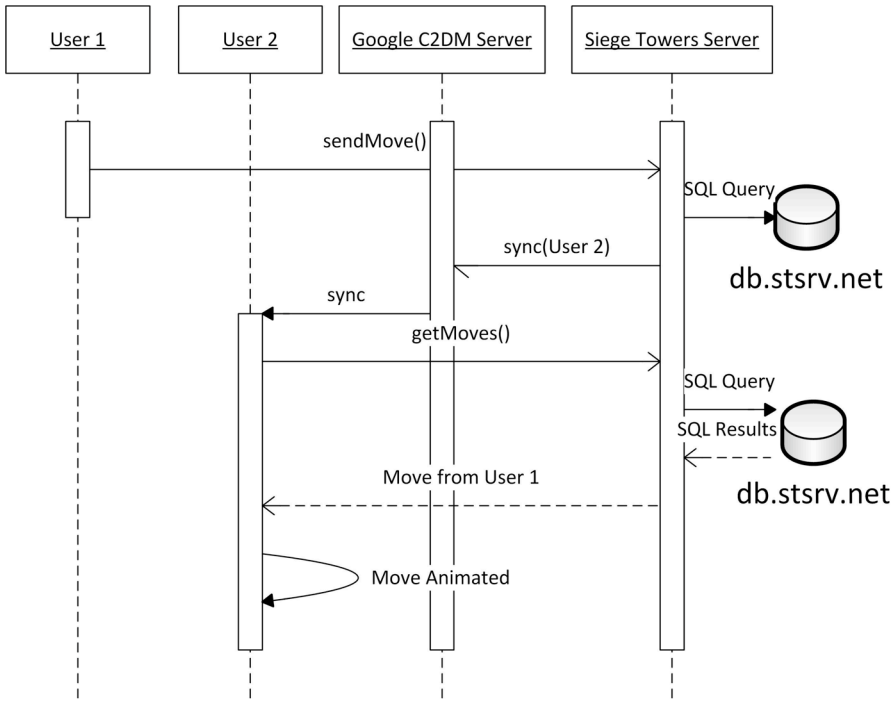
3.0.3.3 Build Sequence



3.0.3.4 Gameplay Sequence



3.0.3.5 C2DM Sequence



3.1 Class SiegeTowersActivity Component

3.1.1 Processing Narrative for SiegeTowersActivity Class

The SiegeTowersActivity is responsible for the initialization of the global GameManager object as well as providing the main menu screen. This main menu displays all current games associated with the user and has a button to launch the NewGameActivity screen.

SiegeTowersActivity represents the actual screen shown at the launch of the application. The class itself is a subtype of a LayoutGameActivity, an AndEngine class that combines a native Android UI and an OpenGL-rendered scene.

3.1.2 SiegeTowersActivity Class Interface Description

The native Android UI is responsible for handling most of the interactions on the main menu screen. This native UI consists of a scrollable ListView, which will contain the current games. The touch events for the ListView are handled by the Android OS. The create new game button will be rendered with AndEngine and will handle touch events by overriding the onClick() method.

3.1.3 Component SiegeTowersActivity Processing Detail

- **Member variables:**
  - **mRm** - ResolutionManager object, gathers information about the user's device screen, including resolution, orientation, center horizontally and vertically. Used in choosing resources to be rendered and setting up the OpenGL environment
  - **mGames** - Reference to ArrayList of Game objects provided by GameManager. Used to populate ListView of games.

3.1.3.1 Design Class hierarchy for SiegeTowersActivity Class

The SiegeTowersActivity class extends the LayoutGameActivitiy class provided by AndEngine. Some methods in the SiegeTowersActivity will override methods in the LayoutGameActivity class.

3.1.3.2 Restrictions/Limitations

The SiegeTowersActivity class is restricted by the hardware limitations of the user's phone in terms of OpenGL rendering capability. It depends on the GameManager object having up-to-date information about the current games.

Performance issues can occur if the user's data connection is slow/unavailable. This will result in the GameManager taking a long time to receive information about the current games. The end result is that the SiegeTowersActivity will not show the game list until they are readily available from the GameManager.

3.1.3.4 Design Constraints

The SiegeTowersActivity must scale and render properly across the supported device resolutions and screen sizes.

3.1.3.5 Processing Detail for each operation of SiegeTowersActivity class

- **launchNewGameActivity()** - Opens the NewGameActivity. Uses Android API to launch.
- **launchGameActivity()** - Opens the TowerBuildingActivity or GamePlayActivity. Uses Android API to launch.
- **refreshListView()** - Updates the ListView with the latest games available from the GameManager. This will be called automatically based on a timer.

3.2 Class NewGameActivity component

The NewGameActivity is the activity that is called from the SiegeTowersActivity in order to invite another user and start a new game.

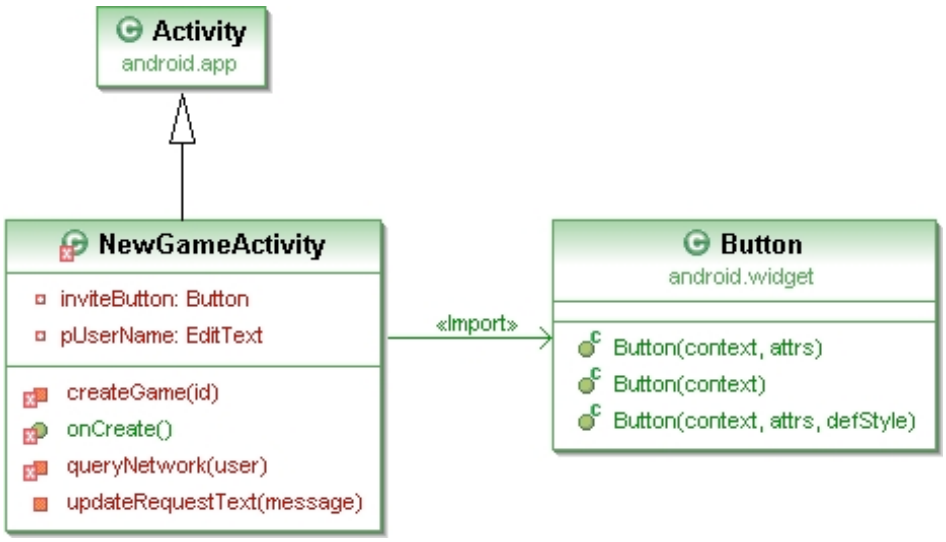
3.2.1 Processing Narrative of Class NewGameActivity

The NewGameActivity component is responsible for requesting a new game. The user types in a phone number of a friend and clicks a button to invite them to play. The activity then queries the network with the phone number, and if it exists in the database creates the new game.

3.2.2 Component NewGameActivity Interface Description

The NewGameActivity screen will be a semi-transparent overlay on the SiegeTowerActivity screen. It will consist of one button that says ‘Invite’ and a textbox to allow the user to type in a phone number. There will also be text in the bottom left corner of the screen after the user clicks ‘Invite.’ It will say ‘Sending Request…’ while the network is queried with the given phone number. If the phone number does not match anything in the database, the text will change to ‘Invalid Request’, allowing the user to input another number. Otherwise, the window closes without changing the text to go back to the menu screen (SiegeTowersActivity).

3.2.3 Component NewGameActivity Processing Detail



- **Member Variables**
  - **inviteButton -** Button that calls userExist() to find out if the username entered exists.
  - **pUserName -** This is the text that is entered by the player, and is the ID sent to the network to check for existence.

3.2.3.1 Design Class hierarchy

The NewGameActivity extends the Activity class provided by Android. The methods in the NewGameActivity will override methods in the Activity class.

3.2.3.2 Restrictions/limitations

The pUserName member variable can only be an alphanumeric string.

3.2.3.3 Performance issues

Performance issues can occur when the user's phone is under intensive use in which the memory modules may not support the memory required for Siege Towers.

3.2.3.4 Design constraints

There are class interactions, so there are many instances in which data is being passed in and manipulated. There are potential issues of variable scope for the data. The classes will need to be implemented so that each class can retrieve the data required for each class's methods

3.2.3.5 Processing Detail for each operation

- **onCreate**: Called by the AndEngine Framework. This method calls the Activity onCreate() method. Listens for the inviteButton click.

- **onClick(View)**: Called by the AndEngine FrameWork. This method will call updateRequestText(int) to update the text in the screen. It also calls queryNetwork(String) to check for the existence of the username.

- **updateRequestText(int id)**: This method takes in a number and changes the text on screen depending on the number. If the number is 0, the text will be changed to 'Sending Request…' and if the number is 1, the text will be changed to 'Invalid Username.'

- **queryNetwork(String pUserName):** Calls the NetworkManager class's method userExist(String userName), sending the variable pUserName. This will return an integer which represents the user ID. If the integer returned is -1, the username doesn't exist, and will be handled by calling updateRequestText(-1).

- **createGame(int id):** Calls the GameManager class's method createGame(int id). It sends the id retrieved in the queryNetwork(String) call. After, it closes this Activity by calling this.finish().

3.2.4 Dynamic Behavior for NewGameActivity Class

When this class is called, a new screen appears that includes a textbox as well as a button labeled 'Invite.' The player will type in a username in the textbox, and click the button when finished. The textbox's contents are saved into the pUserName variable. This variable is sent to the NetworkManager's userExist(String) function, and an integer is returned. This integer is stored in gameID. The NewGameActivity class checks if the gameID equals -1, in which case 'Invalid Username' is printed on the screen and the player can try to input another username. If the gameID is not -1, the GameManager's createGame(int) function is called with gameID. This activity then closes.

3.3 Class TowerBuildingActivity component

The TowerBuildingActivity is the current activity during the build tower phase. This component will handle the user building a tower.

3.3.1 Processing Narrative for component TowerBuildingActivity

The TowerBuildingActivity is responsible for managing the resources during the tower building phase of the game. This class will keep track of the amount of time the user has to construct the user's tower with the amounts of each block type provided by the Game class.

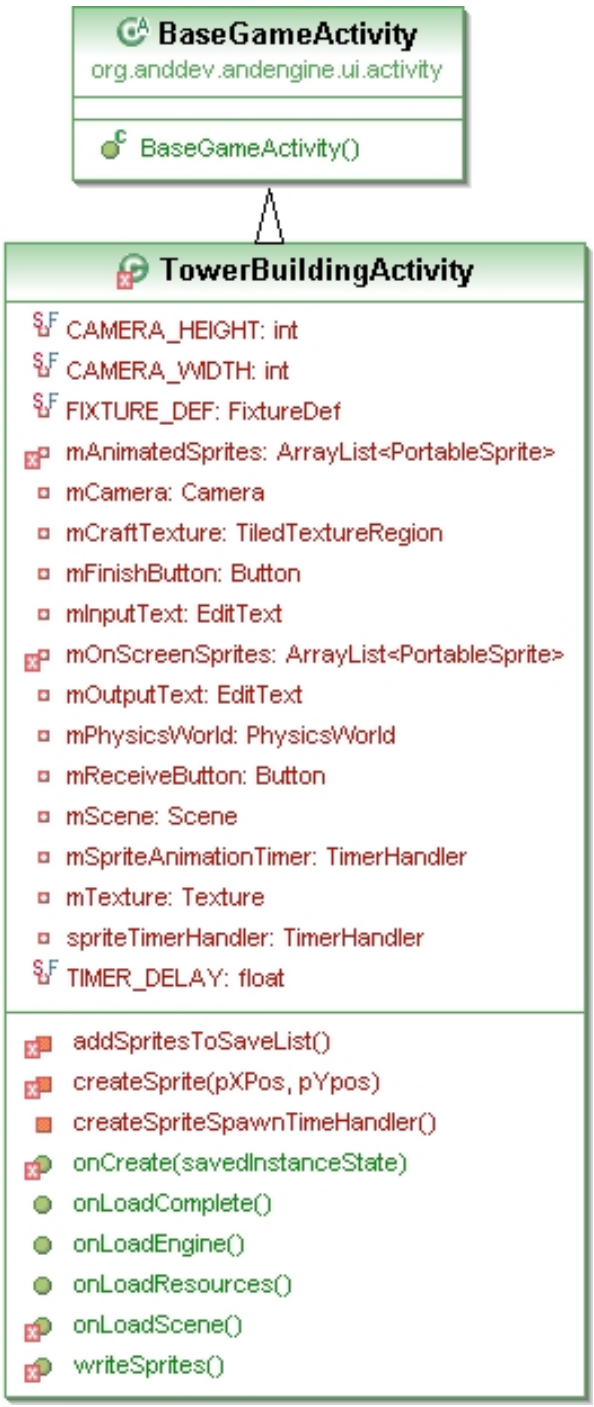
The user is given the ability to create their tower by adding TowerBlocks to the scene. The number of blocks the user can add is dependent on the amount randomly generated by the Game class. The user has the ability to rotate the TowerBlocks by multi-touch features, as well as utilize the recycling bin to remove blocks from the scene.

TowerBuildingActivity represents the actual screen shown during the tower building phase. This Android Activity is a subtype of an AndEngine Activity and it is where the Scene and Physics objects are created. Three BlockGeneratingSprites and all the TowerBlocks sit on the Scene, which is in the TowerBuildingActivity class. BlockGeneratingSprites are defined in section 3.x and TowerBlocks are defined in section 3.x

3.3.2 Component TowerBuildingActivity Interface Description

The interface for input for our entire project is dealt with Android predefined methods. Android will track for touch events, on click events, and on drag events generated by the user. The output will be dependent on what button or sprite the user interacts with in the TowerBuildingActivity.

3.3.3 Component TowerBuildingActivity Processing Detail



- **Member Variables**
  - **mCamera** - Object reference to Camera, the mechanism to control which part of the Scene is displayed on the screen.
  - **mCraftTexture** - Texture to use to create the BlockGeneratingSprites. Holds one or more image to place on these sprites.
  - **mTexture** - Texture to place on the weapon selection sprite.

- **mPhysicsWorld** - The represenation of the Box2d physics engine.
- **mScene** - The object which holds and displays all entities, including sprites.
- **mBody** - the Body object which is the physical represenation in PhysicsWorld for this TowerBlock.
- **mGeneratingSprites** - Array containing the three BlockGeneratingSprites used to create TowerBlocks.
- **mOnScreenSprites** - ArrayList of PortableSprites created from the TowerBlocks currently on the Scene. Used to recreate tower in Game Play Phase.
- **mTimer-** A Countdown timer that contains the time limited for the user to create the tower. The timer class is provided by the Android SDK.

3.3.3.1 Design Class hierarchy

The TowerBuildingActivity extends the BaseGameActivity class provided by AndEngine. The methods in the TowerBuildingActivity will override methods in the BaseGameActivity class.

3.3.3.2 Restrictions/limitations

The TowerBuildingActivity class can only be restricted by the hardware limitations of the user's phone in order to display the game in an intended fashion. Each instance of this class will be unique to each game that is created between two users. (The scope of each instance is limited only between the members of the game they are connected to.)

3.3.3.3 Performance issues

Performance issues can occur when the user's phone is under intensive use in which the memory modules may not support the memory required for Siege Towers as well as the processor speed in order to make calculations required for the physics component and the updating process for the sprites. These constraints will need to be taken into consideration when determining the amount of objects that will be created by the TowerBuildingActivity class to overall maintain stability of the game.

3.3.3.4 Design constraints

There are many class interactions so there are many instances in which data is being passed in and manipulated. There are potential issues of variable scope for the data. The classes will need to be implemented so that each class can retrieve the data required for each class's methods. Potential errors could occur at runtime with array boundaries. They will

need to be considered since the number of resources (blocks and weapons) will be randomly generated.

3.3.3.5 Processing Detail for each operation

- **onCreate**: Called by the AndEngine Framework. This method calls the BaseGameActivity onCreate() method.

- **onLoadEngine**: Called by the AndEngine FrameWork. This method will initialize mCamera and return a new instance of the AndEngine Engine class.

- **onLoadResoruces**: Called by the AndEngine FrameWork. This method creates all textures needed from stored image files.

- **onLoadScene:** Called by the AndEngine FrameWork. This method creates the majority of the TowerBuildingActivity's screen. It will create all three BlockGeneratingSprites plus additional sprites for the color selector, the recycling bin and the done button.

- **onAreaTouched:** The event handler for a touch event on this sprite. Responsible for handling turning physics on or off on this TowerBlock as well as updating it's location on the scene. This is called by the AndEngine framework.

    - **onDoneTouch:** When the done button is pressed, the TowerBuildingActivity will serialize the TowerBlocks to save the state of the user's tower. It will then start a new activity, WeaponsOverlayActivity. This is when the players will choose their weapon distribution. It is defined in more detail in section 3.X.

    - **onBlockColorTouch:** When the blockColor button is pressed, it will update BlockGeneratingSprite type to the corresponding color chosen. It will rotate between the colors red, blue, and green. It will also update what TowerBlock object color will be created when touching a BlockGeneratingSprite.

    - **onRecyclingBinTouch:** When the recyclingBin is pressed, it will set a boolean value of deleteMode to true. When you are in deleteMode, if the user touches a TowerBlock sprite, it will remove the sprite from the scene, decrement the number of

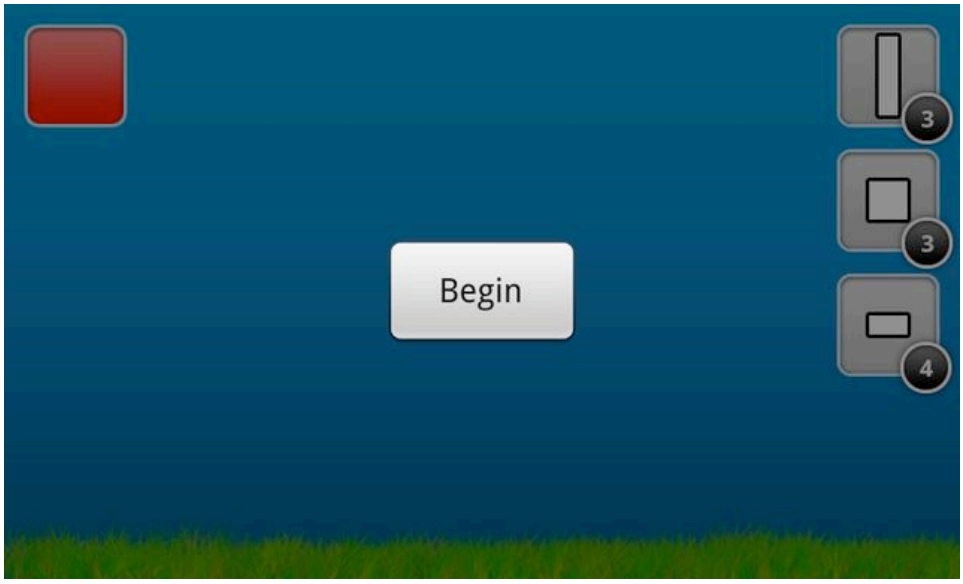
blocks counter, and increment the BlockGeneratingSprite counter that corresponds to that type of block. The scene will still have physics enabled.

- **onBlockGeneratingSpriteTouch**: When any BlockGeneratingSprite is touched and dragged, it will create a TowerBlock that corresponds to the block type of the BlockGeneratingSprite touched. It will add it to the scene, decrement the BlockGeneratingSprite counter, and increment the number of blocks counter.

- **onTowerBlockTouch**: Two seperate actions can occur depending on the boolean value DeleteMode.
    - ***DeleteMode is false***: when a TowerBlock is touched, it will disable physics from the sprite, and will be drag and drop enabled. It will also enable rotating with multi-touch features.
    - ***DeleteMode is true***: when a TowerBlock is touched, it will delete that TowerBlock sprite from the scene, decrement the counter of TowerBlocks and increment the BlockGeneratingSprite counter for that corresponding TowerBlock.

3.4 Description of BlockGeneratingSprite Class

The BlockGeneratingSprite class is used in the Tower Building Phase to create TowerBlock sprites of appropriate size when the user touches and drags within a BlockGeneratingSprite. This class represents the sprites seen on the right hand side of the following mock-up.



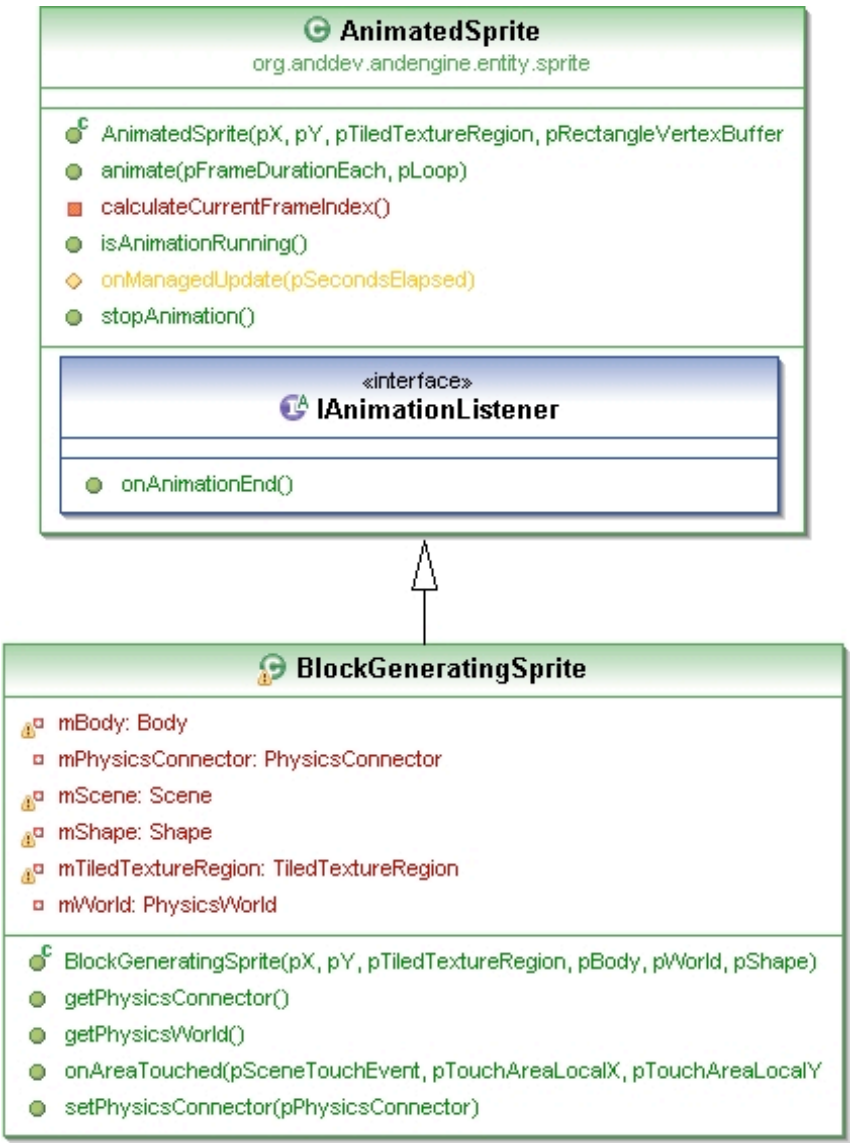
3.4.1 Processing Narrative (PSPEC) for BlockGeneratingSprite Class

The BuildTowerActivity will contain three BlockGeneratingSprites. When a BlockGeneratingSprite is touched and dragged, it will generate a TowerBlock of the corresponding shape type, and attach it to the scene. It will also keep track of how many TowerBlocks of the corresponding shape are available to create.

3.4.2 BlockGeneratingSprite Class Interface Description

This class will handle onTouch events provided by the game engine and Android API. When the user touches a BlockGeneratingSprite, it will output a TowerBlock in the scene that is movable, rotatable, and has been added to the physics environment.

3.4.3 BlockGeneratingSprite Class Processing Detail



Member Variables

- **mWorld** - object reference to PhysicsWorld, the representation of the box2d physics engine.
- **mPhysicsConnector** - object reference to PhysicsConnector which is the mechanism to interface 2D bodies to the PhsyicsWorld.
- **mBody** - the Body object which is the physical represenation in PhysicsWorld for this BlockGeneratingSprite.
- **mShape** - the Shape Enumeration to identify the type of block to generate.
- **mDefenseType** - the Defense Enumeration to identify the defense of the block generated.

3.4.3.1 Design Class hierarchy for BlockGeneratingSprite Class

This class will be instantiated depending on the number of blocks of each type is available to the user. The class will inherit from the AnimatedSprite class.

3.4.3.2 Restrictions/limitations for BlockGeneratingSprite Class

There is a limitation on memory usage of the user's phone. Since the number of blocks will be randomly generated for the user's game. It will need to be tested to see how many instances of the class will be suitable for the game play while maintaining stability between processes on an average user's phone.

3.4.3.3 Performance issues for BlockGeneratingSprite Class

A performance issue will be to intuitively generate a block under the users finger and render the graphics for the block without any delay or errors.

3.4.3.4 Design constraints for BlockGeneratingSprite Class

A constraint for BlockGeneratingSprite is that a TowerBlock is only generated when the user touches and drags their finger from the BlockGeneratingSprite.

3.4.3.5 Processing Detail for each operation of BlockGeneratingSprite Class

- **onAreaTouched** - The event handler for a touch event which occurs in this sprite. Responsible for creating the generation of TowerBlocks in the TowerBuildingActivty Scene. Checks for a touch drag event and creates a TowerBlock of appropriate type. Called by the AndEngine framework.
- **TowerBlock** - Object constructor

3.5 Description of TowerBlock Class

Once the BlockGeneratingSprite class creates the sprite of the particular block type, a TowerBlock will be instantiated as the actual sprite that the user can interact with. The TowerBlock class will also give attributes to the TowerBlock requested for game play such as color, type, and the physical properties of the game environment.

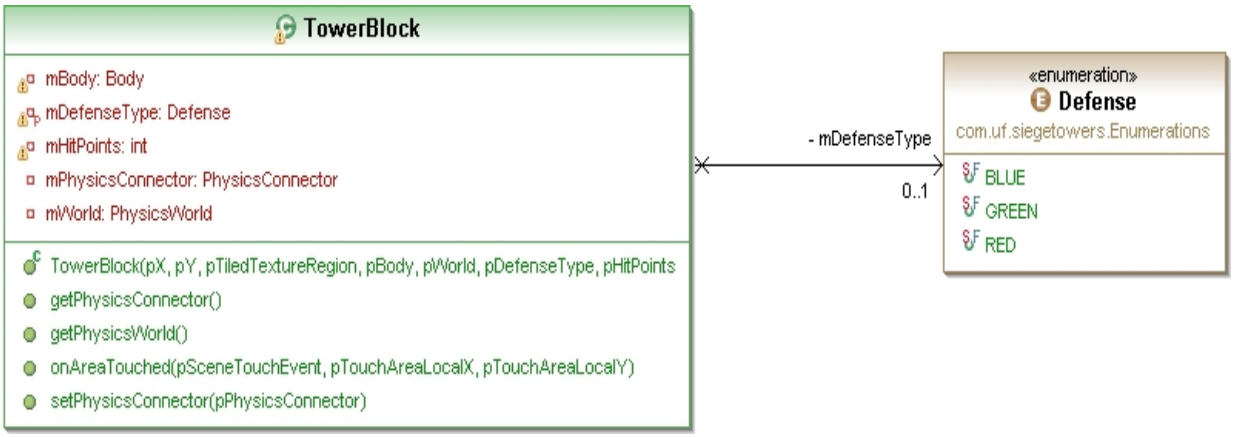
3.5.1 Processing Narrative (PSPEC) for TowerBlock Class

The responsibility of the TowerBlock class is to create an object that the user can utilize for the creation of their tower. It will be called after the user touches and drags off of one of the three block types that are represented by the buttons on the right hand side of the screen. Each TowerBlock created will have contained the attributes that will get passed to the game play phase of the game.

3.5.2 TowerBlock Class Interface Description

This class will handle onTouch events provided by the game engine and Android. The user will input the specified block type requested, as long as that type is available to the user. The output is the block that the user can interact with. The user will also select a color for that specific block type and the input the location of the block once the user releases the block and the physics environment settles the block into place on the screen.

3.5.3 TowerBlock Class Processing Detail



- **Member Variables**
  - **mBody** - the Body object which is the physical represenation in PhysicsWorld for this TowerBlock.
  - **mDefenseType** - Each block will have a defense type of
  - **mHitPoints** - Each block will have 10 health points (damage calculation will be done in game play phase)
  - **mPhysicsConnector** - object reference to PhysicsConnector which is the mechanism to interface 2D bodies to the PhsyicsWorld.
  - **mWorld** - object reference to PhysicsWorld, the representation of the box2d physics engine.

3.5.3.1 Design Class Hierarchy for TowerBlock Class

This class inherits from the AnimatedSprite class that is provided by AndEngine. The AnimatedSprite class contains the methods and the loops to make updates for the sprite drawn on the screen. In addition, the TowerBlock class will call the physicsConnector to enable physics to the sprite that the user can touch.

3.5.3.2 Restrictions/Limitations of TowerBlock Class

The limitation of the TowerBlock class is that an object can only be created on a touch and drag off event of the BlockGeneratingSprite in the TowerBuildingActivity. Only the attributes of each block will be accessible from the build phase and game phase.

3.5.3.3 Performance issues for TowerBlock Class

There could be performance issues when dealing with the rendering of the graphics required for the build tower phase. The graphics rendering will be the most intensive process from this class. The user's phone will need to have the required memory available and the ability to process the graphics for the game work as intended.

3.5.3.4 Design constraints for TowerBlock Class

The TowerBlock class inherits from the AnimatedSprite class to create a tower block that is represented by a sprite and can be oriented by the user with multitouch interaction. The class will only contain information pertaining to that block (color, health points, location coordinates). It focuses on the attributes of the blocks individually, which together form a tower. This class will not manage anything beyond the blocks themselves.

3.5.3.5 Processing Detail for each operation of TowerBlock Class

TowerBlock - Object constructor

getPhysicsConnector/setPhysicsConnector/getPhysicsWorld - accessors

onAreaTouched: The event handler for a touch event on this sprite. It initially disables the physics of the block. It will then call one of two methods. It will call deleteTowerBlock if deleteMode is true. It will call updatePosition if deleteMode is false.

deleteTowerBlock: This method will destroy the block selected by detaching it from the tower building area and

removing it from the ArrayList containing the information and attributes of that particular block.

updatePosition: Will update the position of the block based on the position of the user's touch.

getHealth: Accessor to retrieve health of a block.

setHealth: Sets a constant health integer after the block is placed in the tower building area. It will also be called after damage calculation if an opponent's weapon collides with a particular block.

RotateTowerBlock: The block will be selected by maintaining touch interaction with that block, enlarging the block so that it can be seen over the users finger. There will be a visual circle to show the user that dragging the users finger along the circle can rotate the block.

setDeleteMode: When the recycleButton is pressed, it will enable the user to touch the blocks that the user desires to remove from the tower building area.

3.6 Description of WeaponsOverlayActivity Class

This class will display a user interface that displays the user distribution of weapons. It will handle the user touch inputs to distribute the number of weapons. It will be displayed over the TowerBuildingActivity and blur the background.

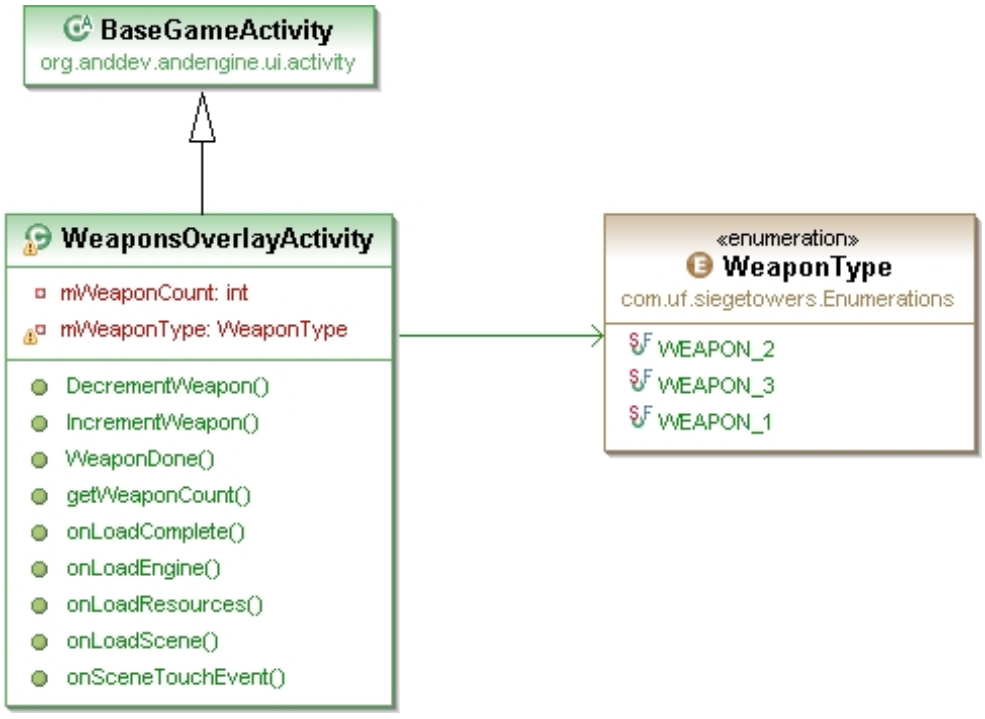
3.6.1 Processing Narrative (PSPEC) for WeaponsOverlayActivity Class

The process of the WeaponsOverlayActivity is to allow the user to distribute the total amount of weapons given by the game manager into the three different types of weapons. Once the BuildTowerActivity is complete (time runs out or both players are done with building their towers), the WeaponsOverlayActivity activity will start.

3.6.2 WeaponsOverlayActivity Class Interface Description

The user will be able to increment and decrement the values for each type of weapon by touching the plus and minus buttons for each type of weapon. Upon touch, the class will display the updated values for the weapons. It will also display the updated value of the number of weapons left to distribute.

3.6.3 WeaponsOverlayActivity Class Processing Detail



3.6.3.1 Design Class hierarchy for WeaponsOverlayActivity Class

The WeaponsOverlayActivity class will get the current game and from the game manager class, it will retrieve the total weapon count created by the game.The user can then select the amount of each weapon the user wishes.

3.6.3.2 Restrictions/limitations for WeaponsOverlayActivity Class

The restriction of this class is that it will only allow the user to modify the amount of each weapon type that the user wants to play the match with. No other data will be changed.

3.6.3.3 Performance issues for WeaponsOverlayActivity Class

No performance issues are present for this class.

3.6.3.4 Design constraints for WeaponsOverlayActivity Class

No design constraints are present for this class.

3.6.3.5 Processing Detail for each operation of WeaponsOverlayActivity Class

- **onPlusTouch:** This will call the incrementWeapon() method for the specific weapon type.

- **onMinusTouch:** This will call the decrementWeapon() method for the specific weapon type.

- **incrementWeapon:** This method will check if there are any remaining weapons to distribute by using the game class method getWeaponCount(). If there are, it will increment the specific weapon by one. It will also decrement the total number of weapons by one.

- **decrementWeapon:** This method will check if there are any remaining weapons in the specific weapon type to decrement by using the game class method getWeaponCount(). If there are, it will decrement the specific weapon by one. It will also increment the total number of weapons by one.

- **onDoneTouch:** It will first call weaponsDone to determine if any weapons are left to distribute. It will then check to see if opponent is ready. If ready, both players will move to the game play activity.

Else, the user will be automatically sent back to the menu screen until a notification tells the user the opponent is ready to start the match.

- **weaponsDone:** This method will first check if the user has any remaining weapons to distribute. If so, it will notify the user that they have remaining weapons to distribute. It will return a boolean value.

3.7 Description for GamePlayActivity Class

The GamePlayActivity Class represents the projectile that is shot at the opponents tower. It will contain all the information about the projectile and will handle collision events.

3.7.1 Processing Narrative (PSPEC) for GamePlayActivity

This class is responsible for all actions concerning the projectile that is fired at the opponent's tower. This includes knowing the weapon type fired. Additionally it will need to be able to change its weapon type as the user interacts with the color selector button and selects the color. The class will also need to handle the event when it is touched and dragged in the slingshot as the user aims and prepares to fire. The user can zoom and pan around the scene.

Figure - 3.7.1 Overall Scene of GamePlayActivity

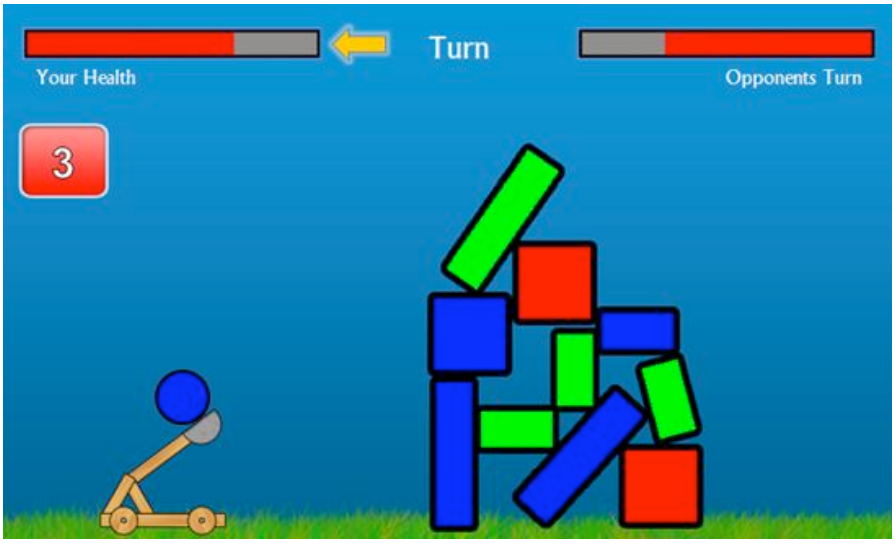
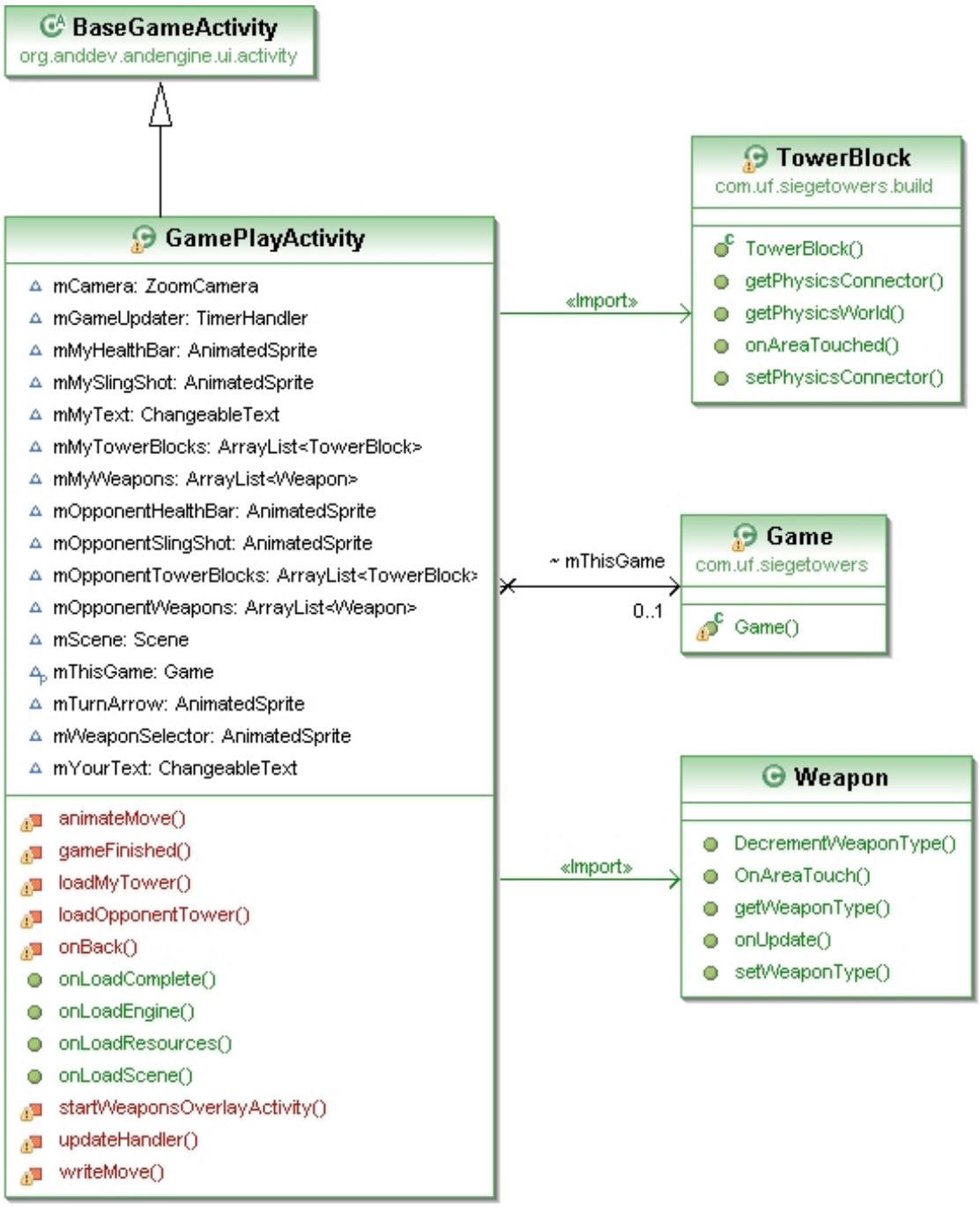


Figure 3.7.2 - Overlayed Entities Fixed to Camera

3.7.2 GamePlayActivity interface Description

The GamePlayActivity Class takes a string when it is launched which refers to a particular Game object. The GamePlayActivity then request from the GameManager the reference to the Game object. This class also takes as inputs two serialized files, which represent the collection of sprites that form this player and the opponent's tower.

3.7.3 GamePlayActivity Processing Detail



Member Variables

mMyTowerBlocks - The TowerBlocks which form this player's tower.
mOpponentTowerBlocks - The TowerBlocks that form the opponent's tower.
mMyWeapons - The ArrayList of Weapons available to this user.
mOpponentWeapons - The ArrayList of Weapons available to the opponent
mThisGame - The Game object which represents the state of this game
mWeaponSelector - The sprite on the top left portion of the Game Play Screen camera view that selects the next type of weapon to launch (if available).
mMySlingShot - The representation of the object which launches this player's weapons

mOpponentSlingShot - The representation of the object which launches the opponent's weapons
mCamera - The ZoomCamera which controls the view of the scene on the screen. Controls zooming in and zooming out the camera on the scene.
mGameUpdater - The TimerHandler which updates the state of the GamePlayActivity with the information reflected in the Game object.
mTurnArrow - The Arrow which is attached to the camera that indicates which player is due for the next move.
mMyHealthBar - The bar representing the ratio of damage incurred by this
mOpponentHealthBar - The bar representing the ratio of damage incurred by the opponent.
mMyText - The text on the left hand side of the camera view.
mOpponentText - The text on the right hand side of the camera view.

3.7.3.1 Design Class hierarchy for GamePlayActivity

This class inherits from the BaseGameActivity Class an AndEngine class. The functionality offered by this class includes orientation sensing, method handlers when the engine and the scene loads and access to the accelerometer. The source code for this class can be found online at googlecode.com/andengine.

3.7.3.2 Restrictions/limitations for GamePlayActivity

The GamePlayActivity is only to be launched when the Game object passed to it on launch is in the GAMEPLAY state.

3.7.3.3 Performance issues for GamePlayActivity

The GamePlayActivity Class must be able to support the Box2d physics engine extension of Andengine.

3.7.3.4 Design constraints for GamePlayActivity

This design is dependant on updates from the GameManager to keep the status of the game updated for display to the user.

3.7.3.5 Processing Detail for each operation of GamePlayActivity

onLoadComplete() - Called by the AndEngine Framework
onLoadEngine() - Called by the AndEngine Framework and is responsible of initializing mCamera to allow for touch panning and mutli touch pinch-zoom.
onLoadResources() - Called by the AndEngine Framework and is responsible for creating the Texture objects that will be used to overlay images onto TowerBlocks, WeaponBlocks and the slingshot sprites.
onLoadScene() - Called by the AndEngine Framework and is responsible for placing the mWeaponSelector sprite, displaying the towerblocks, as well as health bars and turn notification.

onBack() - Called by the Andriod OS to handle the hardware back button press event. This method will save the state of this game and load the main menu.
loadMyTower() - Loads the tower file specified in the Game object for this player. This method translates the coordinates of the TowerBlocks to allow this player's tower to be constructed on the left hand side of the scene.
loadOpponentTower() - Loads teh tower file specified in the Game object for the opponent. This method translates the coordinates of the TowerBlocks to allow for this player's tower to be constructed on the right hand portion of the scene.
updateHander() - Updates the GamePlayActivity with data from the Game object.
gameFinished() - Called upon completion of the game. Is responsible for closing the GamePlayActivity and launching the GameStatActivity class to display the game statistics.
writeMove() - called to write this player's Move object to the file system.
animateMove() - called to read the opponents Move object from the file system and replay the opponents player move.

3.8 Description for Weapon Class

The Weapon Class represents the projectile that is shot at the opponents tower. It will contain all the information about the projectile and will handle collision events.

3.8.1 Processing Narrative (PSPEC) for Weapon Class

This class will be responsible for all actions concerning the projectile that is fired at the opponent's tower. This includes knowing what weapon type of weapon it is. Additionally it will need to be able to change its weapon type as the user interacts with the BuildGameActivity and selects the color. The class will also need to handle the event when it is touched and dragged in the slingshot as the user aims and prepares to fire.

3.8.2 Weapon Class interface description

The weapons class will need to handle both input and output. The input will occur as the user clicks and drags on the weapon picture. The Weapon will need to determine the angle of drag and the distance to determine the force to apply to the Weapon on launch. Other input includes the selected weapon type by the user when choosing the color projectile to shoot. The Weapon will handle by changes it's internal weapon type.

Output from the weapon will occur when the game queries the weapon for it's weapon type. The weapon will return its color.

3.8.3 Weapon Class Processing Detail

Member Variables
mWeaponType - The weapon type for this weapon of type enumeration

3.8.3.1 Design Class hierarchy for Weapon class

This method does not inherit from any parent classes. It will contain all it's own functionality and information.

3.8.3.2 Restrictions/limitations for Weapon class

The weapon class will only be of the specified 3 types of weapons. Also there will only be one visible weapon on the screen at a time that the user can interact with.

3.8.3.3 Performance issues for Weapon class

There may be performance issues when the weapon is fired in an unusual manner. AndEngine will be utilized for collision detection and physics calculations. The performance limitations of the engine also apply to this Weapon Class.

3.8.3.4 Design constraints for Weapon class

The design is implemented to encapsulate the functionality and information of the Weapon object displayed in the GamePlayActivity. No additional processing or capabilities are required or will be designed.

3.8.3.5 Processing Detail for each operation of Weapon class

decrementWeaponType : reduces the number of weapons that the Game class has. This method is called after a collision and the weapon is being removed from the game.

OnAreaTouch : Handles the touch and drag aiming the user will perform when firing at the opponent's tower.

getWeaponType : Returns the weapon type of this Weapon. The weapon type is an enumeration.

onUpdate : will refresh the type of weapon and determine if the weapon needs to be decremented.

setWeaponType : sets this weapon to the type specified by the passed in enumeration variable.

3.9 Description of Game Class

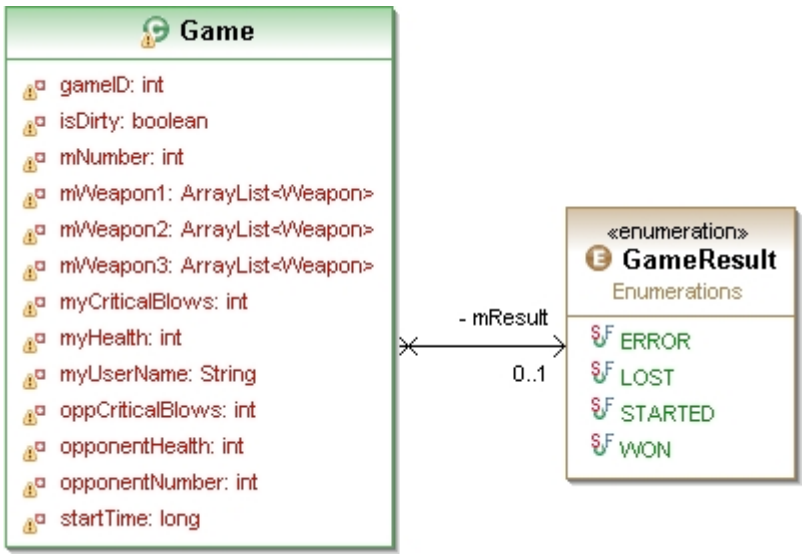
The Game class stores information about a single game being played. The game class will contain all necessary information to represent a game.

3.9.1 Processing Narrative (PSPEC) for Game class

The game class will be a data structure intended to describe that state of a particular game. Multiple game objects can exist in a situation where the user is playing multiple games at once.

3.9.2 Game class interface description

Since the game class is only a data structure the only interface with it will be accessor methods. All members will be modifiable. Any object containing a game will be able to modify all states of that game.



3.9.3 Game class Processing Detail

Member Variables
gameID - a unique number to identify the particular game
isDirty - A boolean specifying if the game needs to be updated on the network
myNumber - The phone number of the user playing the game
mWeapon1, mWeapon2, mWeapon3 - an ArrayList containing all of the weapons of each particular type
myCriticalBlows - The cumulative number of critical hits made by the user during this game
myHealth - The remaining health of the user
myUserName - the unique name of the user
oppCriticalBlows - The cumulative number of critical hits made by the opponent during this game
opponentHealth - The remaining health of the opponent
opponentNumber - The phone number of the opponent playing this game
startTime - The time stamp of when the game began

3.9.3.1 Design Class hierarchy for Game class

This method does not have any parent or children classes. The game class will contain all of the necessary data and functionality.

3.9.3.2 Restrictions/limitations for Game class

The game class is restricted to containing information for a single game. A containing class will need to keep a list of multiple game instances to represent multiple simultaneous games.

3.9.3.3 Performance issues for Game class

The game class is a very small data structure with limited attributes. There are no expected performance issues with this class.

3.9.3.4 Design constraints for Game class

The Game class is designed as a Data Structure and will contain no additional methods other and accessors.

3.9.3.5 Processing Detail for each operation of Game class

This class will only contain accessor and will not have any additional methods.

3.10 Description of ResolutionManager Class

The ResolutionManager class gathers and computes information about the target device's display properties. It contains fields for the display width and height (in landscape orientation), horizontal and vertical center values, and a resolution name string (QVGA, HVGA, VGA, etc.). These values are used in the implementation and scaling of UI elements for different size screens.

3.10.1 Processing Narrative for ResolutionManager Class

The resolution manager accesses the Android API's Window Manager to retrieve and store the user's display properties. It stores the width and height and uses these values to calculate horizontal and vertical center values, as well as a resolution name string.

3.10.2 ResolutionManager Interface Description

Member Variables
mWidth - Width of screen in landscape mode (pixels)
mCenterY - Vertical middle of screen (pixels)
mCenterX - Horizontal middle of screen (pixels)
mHeight - Height of screen in landscape mode (pixels)
mRes - String representation of resolution (QVGA, HVGA, VGA, etc.)

3.10.3.1 Design Class Hierarchy for ResolutionManager Class

This method does not inherit from any parent classes. It will contain all it's own functionality and information.

3.10.3.2 Restrictions/Limitations for ResolutionManager Class

There are no restrictions/limitations of the ResolutionManager class. It relies on an Android API call, which exists on all target Android devices.

3.10.3.3 Performance Issues for ResolutionManager Class

There are no performance issues for ResolutionManager, as its only calculations are integer divisions.

3.10.3.4 Design Constraints for ResolutionManager Class

The ResolutionManager class is designed as a data structure and will contain no additional methods or accessors.

3.10.3.5 Processing Detail for Each Operation of ResolutionManager Class

The constructor will retrieve the screen parameters and calculate the center values and the string representation.

3.10.4 Dynamic Behavior for ResolutionManager Class

The ResolutionManager is a static class.

3.10.4.1 Interaction Diagrams for ResolutionManager Class

The only interactions with this class are through activity classes as a data structure

3.11 Description for NetworkManager Class

The NetworkManager class contains a set of methods used by the rest of the Siege Towers client application. This class uses the org.apache.http library (produced by the Apache Foundation) to communicate across the network to the Apache server; all communication with the server will be via HTTP packets. POST messages will be used to upload information, and GET messages will be used to retrieve information.

3.11.1 Processing Narrative (PSPEC) for NetworkManager Class

Siege Towers clients will store game state information on the server, and will also send Towers and Moves (serialized Java objects) across the network; this will be accomplished by a messaging framework encapsulated within the NetworkManager class. Activities will be able to asynchronously call update and retrieval methods on the NetworkManager whenever the functionality is needed.

Every Java method in the NetworkManager class will have a corresponding PHP script on the server, with the same name. The PHP script will accept the GET or POST request, execute the appropriate SQL command, and return the results.

3.11.2 NetworkManager Class Interface Description

The interface will be a simple set of public methods, which return integers to indicate error status. Parameters, which need to be modified, will not be returned; instead they will accept references to the Game objects.

3.11.3.1 Design Class hierarchy for NetworkManager Class

The NetworkManager class will be a single instantiated Java object inheriting directly from Object (no explicit inheritance.) It will be composed of several objects from the org.apache.http library.

3.11.3.2 Restrictions/limitations for NetworkManager Class

There should only be a single instance of the NetworkManager class at any point in the application. If several NetworkManager classes were running the game state could be altered in unpredictable ways. This will be accomplished by having the NetworkManager instantiated as a single static instance.

3.11.3.3 Performance issues for NetworkManager Class

There are no expected performance issues with this class. It is a set of very simple methods which use Apache's org.apache.http library to send small serialized objects and integers.

3.11.3.4 Design constraints for NetworkManager Class

The NetworkManager class is intended to act as an architecture layer between the network and the user interface activities. Therefore the NetworkManager will never interact directly will the user or will have no visual components.

3.11.3.5 Processing Detail for each operation of NetworkManager

int createGame(Game pGame) Accepts a Game object reference from the Game Manager. Makes a POST call to the corresponding PHP script, createGame.php, which will create a new entry in the Game table. Returns the unique Game ID.
int updateGame(Game pGame) Accepts a Game object reference from the Game Manager. Makes a call to the corresponding PHP script, updateGame.php, which will retrieve the current game state for the opponent and the creator. Modifies the Game object.
int userExist(String username) Accepts a String which contains the user name which is being searched for. Makes a GET call to the corresponding PHP script, userExist.php, which will execute a SQL query and return the unique ID of the user with that user name. Returns -1 if the user is not found.
ArrayList<Game> getCurrentGames() Called when the game loads. Makes a GET call to the corresponding PHP script, getCurrentGames.php, which queries the database to get a list of all current games. The Game objects will be created and placed in an ArrayList. The PHP script will need a user ID, but no parameter is needed in the method call because it will be clear from the context which user is being queried; the user will always be the owner of the phone.

3.12 Description for C2DMHandler Class, as a Utility Service

3.12.1 Advantages of this class's implementation

Out of the many choices for keeping our application up to date with the data that resides on the server side (and thus streamlining the experience for the user), we decided upon the use of Cloud to Device Messaging (henceforth denoted as C2DM). This new feature was introduced in Android 2.2 by Google, who also provide a simple, yet comprehensive API for implementation. C2DM is implemented using "Push" technology. This means that instead of the Android device "ping"ing a server at set intervals in order to ascertain whether or not it needs to update, the Google C2DM server will actually "push" a notification to the individual phone whenever instructed to do so by the registered application server in the cloud. This methodology will prove very efficient, especially in terms of power consumption, as we do not

need to have processes running in the background of the Android OS in order to make sure that our user stays up to date. Also, this means that if a user chooses not to utilize our application for certain periods of time, it will not needlessly be devouring precious memory or power.

3.12.2 When and How C2DM will be used in our Application

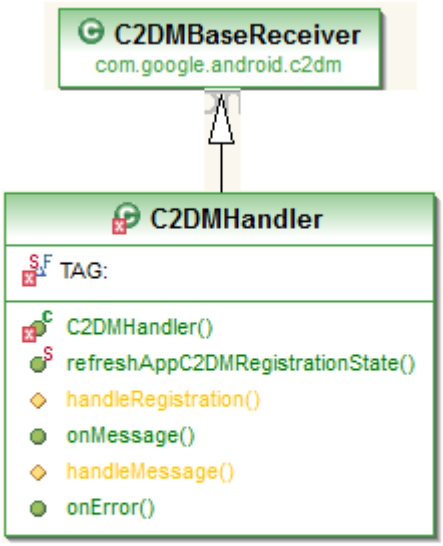
Our goal is for the C2DM to run seamlessly in the background of the application, so that the user need not concern himself with the configuration or workings of any part of the C2DM process. Thus, there are three major instances where C2DM will be implemented within our application.

1. Upon the first run of the application after installation. During the first run of the application, the application will collect the necessary Google account information from the user's phone (or ask for them if need be). Then the application will create a new Intent to register the application and device with the Google C2DM server. After receiving the Registration ID from the Google server, the application will post this information to the APACHE application server for storing within the database.
2. Naturally, whenever the APACHE server's database has received an update concerning a specific user, the application will notify the Google C2DM server to push a notification to the phone. The phone will receive this Intent from the C2DM server and will then process this within the C2DMReceiver class and call the necessary methods to synchronize with the APACHE server's database. Also, the Google C2DM server will occasionally send the application an updated Registration ID. Our application will be able to decipher the type of message received so that it will know to either update its Registration ID with the APACHE server or to request a synchronization of data with the APACHE server.
3. Our application will contain a manual switch for the user to manipulate within the Options menu. This will allow the user to disable or enable push notifications at will. If the user turns notifications on, then the same process will occur as if the phone was starting up for the first time as described above. If the user turns off push notifications, then the application will automatically create a new Intent to unregister the application from the Google C2DM server.

3.12.3 Design Class hierarchy for C2DMHandler Class

Classes and Methods for C2DM:
On the Android Client side, there is one primary class that handles C2DM protocols, which contains a host of methods which will be able to decipher the received Intents from the C2DM server as well as output communications to the C2DM server for registration purposes. For the sake of coherence and avoiding ambiguity, this document will break down the C2DMHandling class into its integral components and then describe them in detail, starting with the class declaration itself.

The C2DMHandler class must extend the C2DMBaseReceiver class provided by the Google API (com.google.android.c2dm.C2DMBaseReceiver), which is imported.



3.12.4 Restrictions and Limitations for C2DMHandler Class

The C2DMHandler should only be called upon when needed in the three instances described above in Section 3.x.2.

3.12.5 Performance issues for C2DMHandler Class

The speed of performance for the handling of C2DM will be limited by the response latency of the Google C2DM server. Google makes no guarantees for immediate delivery of C2DM pushes, but on average they report response times in terms of seconds. At worst, the user should see no more than a two minute delay between turns.

3.12.6 Design constraints for C2DMHandler Class

The C2DMHandler class should act entirely in the background and never noticed by the user. All RegistrationID requests and synchronizations with the server should happen without any impact to the running of the application or of any other application that the user should be running. The user can only influence C2DM via the Push Notification Enable button in the Options menu.

3.12.7 Processing Detail for each operation of component methods Class C2DMHandler

- This class will import the following:
    - import android.accounts.Account;

    - import android.accounts.AccountManager;
    - import android.content.ContentResolver;
    - import android.content.Context;
    - import android.content.Intent;
    - import com.google.android.c2dm.C2DMBaseReceiver;
    - import com.google.android.c2dm.C2DMessaging;

- The C2DMHandler class must extend the C2DMBaseReceiver class provided by the Google API (com.google.android.c2dm.C2DMBaseReceiver) which is imported.
- This class can be called by the Android OS once it receives the Context and Intent pushed to the phone by the Google C2DM servers.
- This class can be called by the application in order to make Registration and Unregistration requests.
- This class's primary function is to be able to take Intents and Contexts related to C2DM and process it appropriately. Upon receiving a push notification it must be able to determine whether or not the C2DM server is attempting a registration update or is pushing a sync message to the application. If the application is the caller of this class and its methods, then it must be able to handle requests corresponding to registration and unregistration of the user's application from the C2DM server.

Public void onError(Context context, String ErrorId) {}

- If for any reason an error occurs when the C2DMessaging class sends a REGISTER post to the C2DM server, then the C2DM server will push back an error Intent. The error is stored as a string and then read in as a parameter for this method. The table below details the possible error strings and their meanings.

Error Code	Description
SERVICE_NOT_AVAILABLE	The device can't read the response, or there was a 500/503 from the server that can be retried later. The application should use exponential back off and retry.
ACCOUNT_MISSING	There is no Google account on the phone. The application should ask the user to open the account manager and add a Google account. Fix on the device side.
AUTHENTICATION_FAILED	Bad password. The application should ask the user to enter his/her password, and let user retry manually later. Fix on the device side.
TOO_MANY_REGISTRATIONS	The user has too many applications registered. The application should tell the user to uninstall some other applications, let user retry manually. Fix on the device side.
INVALID_SENDER	The sender account is not recognized.
PHONE_REGISTRATION_ERROR	Incorrect phone registration with Google. This phone doesn't currently support C2DM.

public void onMessage(Context context, Intent intent)

- The key method within the C2DMReceiver class.
- Whenever a push notification is received, the application will call this method, and it is the responsibility of this method to ascertain the nature of the push notification.
- Push notification Intents are packaged with “actions” that will detail their purpose, and only simple checks made upon these actions via the getAction() method are needed to ascertain the purpose of the Intent.
- ```
  if(intent.getAction().equals("com.google.android.c2dm.intent.REGISTRATION")) {
        handleRegistration(context, intent);
    }
  else if(intent.getAction().equals("com.google.android.c2dm.intent.RECEIVE")) {
        handleMessage(context, intent);
     }
  ```

protected void handleRegistration(Context context, Intent intent)

- This method is called if the notification received by the application passes the check:
    - if(intent.getAction().equals("**com.google.android.c2dm.intent.REGISTRATION**"))

- The main functionality of this method is to take the received RegistrationID from the Intent passed in as a parameter. This is done by the follow code:
    - String registration = intent.getStringExtra("**registration_id**");

- The method must also check for registration errors and if an error is found, then this method must call the onError() method.
    - if (intent.getStringExtra("**error**") != null)

- After retrieving the RegistrationID, this method will then update our APACHE application server’s database via HTTP post and passing the RegistrationID as a parameter in that post.

protected void handleMessage(Context context, Intent intent)

- The handleMessage method will act similarly to the handleRegistration method.
- This method is called after the check:
    - if(intent.getAction().equals("**com.google.android.c2dm.intent.RECEIVE**"))

- This method will pull from the intent the message contents, which are stored as “extras” and accessed via the getExras() method.
- Regardless if the device is idle or not, this method will then send a synchronization request to the APACHE server to request the updated information via HTTP post. This will happen in the background without prompt to the user.
- In order to do this, this method will call the NetworkManager class’s updateStates() method in order to synchronize the application.

Public static void refreshAppC2DMRegistrationState(Context context)

- This method is what provides the functionality of the Push Notification button in the Options menu. Depending on whether the user turns the switch on or off, this method will either register the application with the Google C2DM server or unregister it.
  ```

- if (autoSyncDesired == true) {
  C2DMessaging.register(context, Config.C2DM_SENDER); }
  else { C2DMessaging.unregister(context); }

3.13 Role of the Siege Towers Application Server

3.13.1 How the Application Server Sends Messages

This section describes how the APACHE application server sends messages to the Google C2DM server so that it can be forwarded to the Android client application running on a mobile device. Before the APACHE application server can send a message to an Android application, it must have received a Registration ID from it. In addition, the APACHE server must be authorized by Google to send push messages from the cloud to the Google C2DM server. To send a message, the application server issues a POST request to https://android.apis.google.com/c2dm/send that includes the following as parameters:

Field	Description
registration_id	The registration ID retrieved from the Android application on the phone. Required.
collapse_key	An arbitrary string that is used to collapse a group of like messages when the device is offline, so that only the last message gets sent to the client. This is intended to avoid sending too many messages to the phone when it comes back online. Note that since there is no guarantee of the order in which messages get sent, the "last" message may not actually be the last message sent by the application server. Required.
data.<key>	Payload data, expressed as key-value pairs. If present, it will be included in the Intent as application data, with the <key>. There is no limit on the number of key/value pairs, though there is a limit on the total size of the message. Optional.
delay_while_idle	If included, indicates that the message should not be sent immediately if the device is idle. The server will wait for the device to become active, and then only the last message for each collapse_key value will be sent. Optional.
Authorization: GoogleLogin auth=[AUTH_TOKEN]	Header with a ClientLogin Auth token. The cookie must be associated with the ac2dm service. Required.

3.13.2 Possible Response Codes:

The Google C2DM server can send back a variety of response codes to the APACHE application server that it must be able to handle.

Response	Description
200	Includes body containing: ● id=[*ID of sent message*] ● Error=[*error code*] ○ QuotaExceeded — Too many messages sent by the sender. Retry after a while. ○ DeviceQuotaExceeded — Too many messages sent by the sender to a specific device. Retry after a while. ○ InvalidRegistration — Missing or bad registration_id. Sender should stop sending messages to this device. ○ NotRegistered — The registration_id is no longer valid, for example user has uninstalled the application or turned off notifications. Sender should stop sending messages to this device. ○ MessageTooBig — The payload of the message is too big, see the limitations. Reduce the size of the message. ○ MissingCollapseKey — Collapse key is required. Include collapse key in the request.
503	Indicates that the server is temporarily unavailable (i.e., because of timeouts, etc). Sender must retry later, honoring any Retry-After header included in the response. Application servers must implement exponential back off. Senders that create problems risk being blacklisted.
401	Indicates that the ClientLogin AUTH_TOKEN used to validate the sender is invalid.

4 User interface design

4.1 Description of the user interface

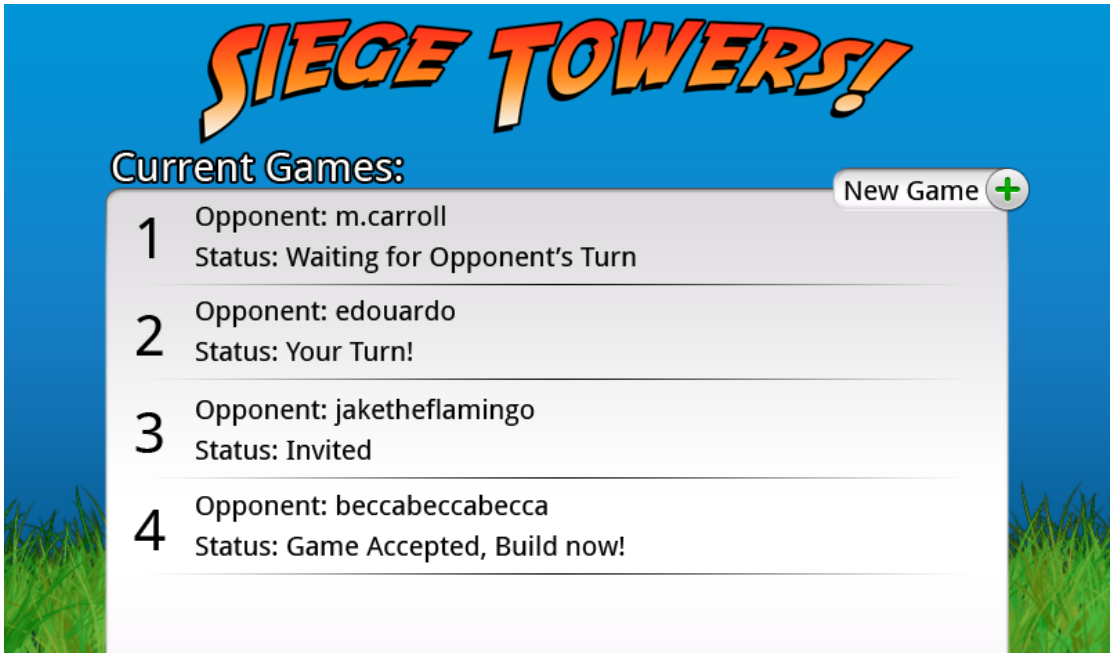
The user interface will be split up into 4 major activities: SiegeTowersActivity, TowerBuildingActivity, WeaponsOverlayActivity, and the GamePlayActivity. The user will be able to navigate through these screens to reach the components of the game that requires the user's input.

4.1.1 Screen images

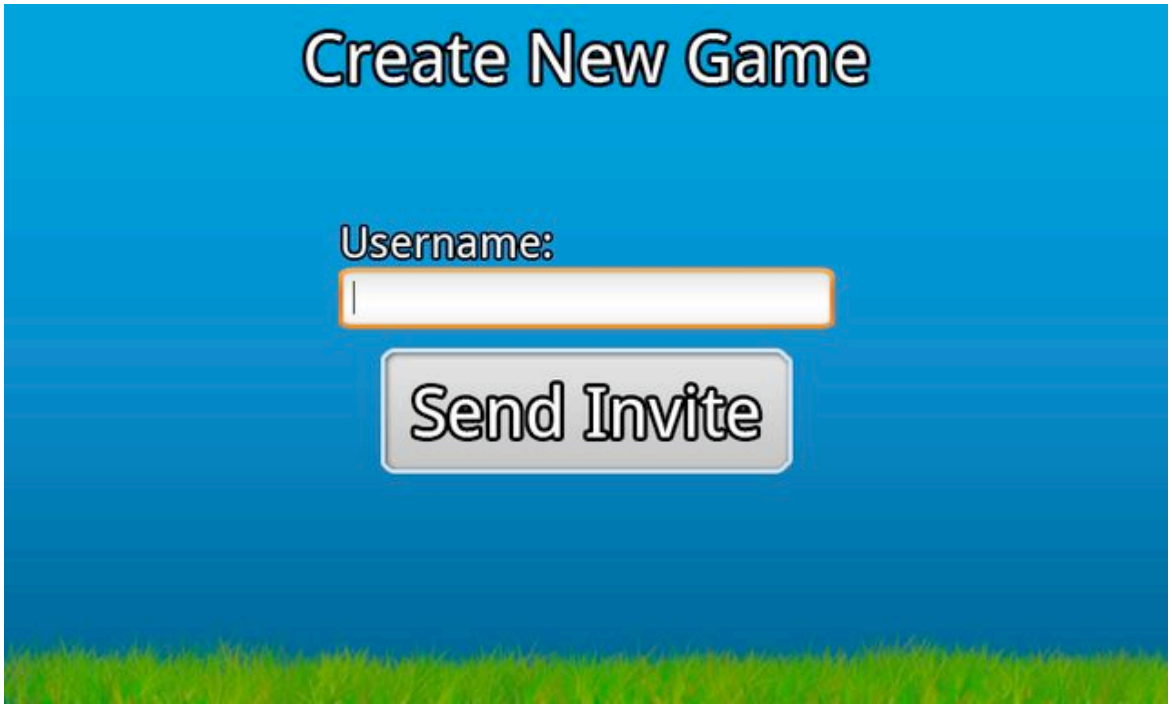
**Note: These screen mock-ups were designed for the SDD. They may not be the final UI design for the game.*

Opening Screen

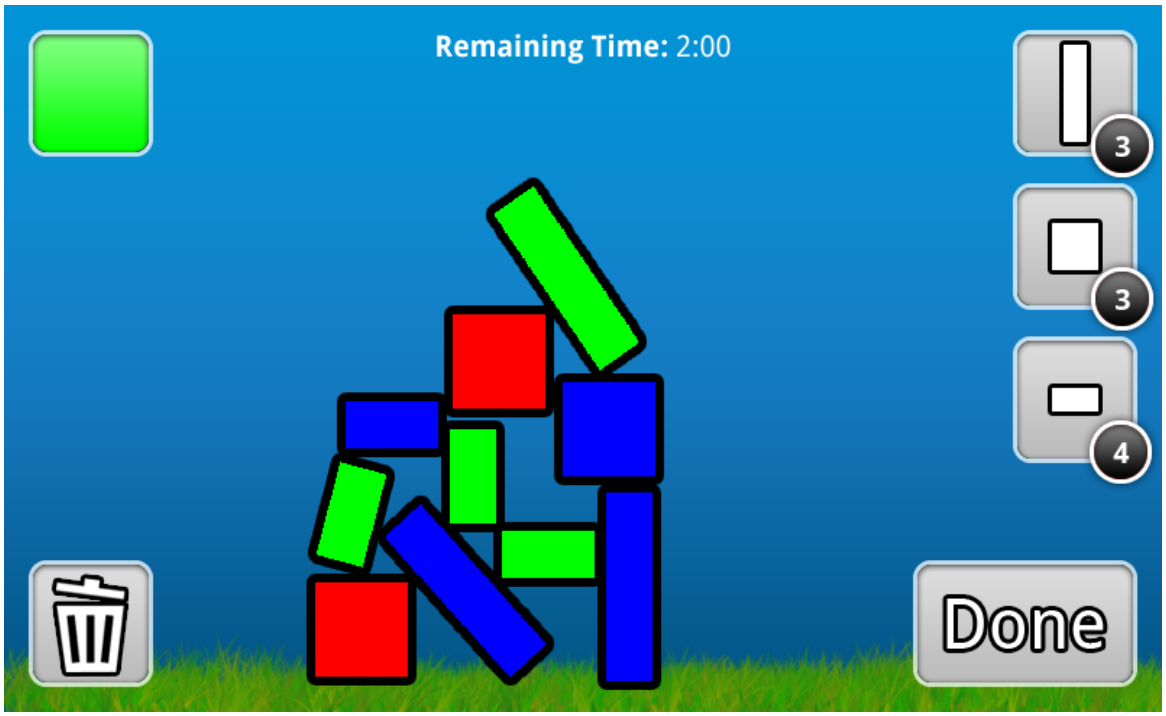
Game Selection Menu



Create New Game



Tower Building Screen



Weapons Distribution Overlay

Game Play Screen

4.1.2 Objects and actions

Each Screen interface will have buttons using a OnTouchListener() in order to navigate menus. The Hardware button “Back” will allow the user to go back to a previous menu or to the home menu depending on which state the user is in. The user will be able to use alphanumeric keys to enter information in the text boxes for registration on the user’s device.

The onTouchListener() for the list of games will enable the user to add a new game, select an opponent to play against, or touch a game on the list to continue.

During the Tower building phase, the user will be able to use the touch screen to drag in blocks to make the user’s tower. The recycleBin button will put the tower into delete

mode. The user can then pick blocks to delete from the area by touching them and retouching the recycle bin button to go back into build mode. In addition to dragging in a block from the types that are available, the user can rotate the block selected by moving another finger around the block (using multitouch) before setting it in place on the area. The user will also be able to touch the color selection button to change color of the block before dragging it onto the tower building area. The user has the option to touch the done button or allow the time to run out in order to progress to the next stage of the game.

Once the time is done, the players will then be able to set the amount of each weapon type by touching the plus and minus buttons, provided that the total weapons amount available is greater than zero. The player will touch the done button to exit the overlay.

The player will then progress to the game play screen, which upon the user's turn, the user will be able to set the weapon type desired. Once that is complete, the user will touch the weapon loaded in the launcher and pull back in order to shoot the projected directed towards the opponent's tower. Damage will be calculated for each tower and when the players are out of weapons, the game will determine the winner. Stats will be displayed on the screen at the end.

4.2 Interface Design Rules

Ben Shneiderman's "Eight Golden Rules of Interface Design" provide the most clear standards to use in developing the interface. The eight rules are as follows:

1 Strive for consistency
Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent commands should be employed throughout.

2 Enable frequent users to use shortcuts.
As the frequency of use increases, so do the user's desires to reduce the number of interactions and to increase the pace of interaction. Abbreviations, function keys, hidden commands, and macro-facilities are very helpful to an expert user.

3 Offer informative feedback.
For every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.

4 Design dialog to yield closure
Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and an indication that the way is clear to prepare for the next group of actions.

5 Offer simple error handling.
As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should be able to detect the error and offer simple, comprehensible mechanisms for handling the error.

6 Permit easy reversal of actions.
This feature relieves anxiety, since the user knows that errors can be undone; it thus encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.

7 Support internal locus of control.
Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.

8 Reduce short-term memory load.
The limitation of human information processing in short-term memory requires that displays be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

4.3 Components available

The user interface consists of the following components:

- SiegeTowersActivity *(Section 3.x)*
- NewGameActivity *(Section 3.x)*
- TowerBuildingActivity *(Section 3.x)*
- WeaponsOverlayActivity *(Section 3.x)*
- GamePlayActivity *(Section 3.x)*

Each of these screens inherits from the Activity class and will be used to create the various application screens and menus. They are all described in greater detail in

Section 3 of this document. The provided Android widgets are used as well as AndEngine features.

4.4 UIDS description

No user interface design system is to be used in the development of SiegeTowers.

5 Restrictions, Limitations, and Constraints

The restrictions and constraints of Siege Towers shall be listed with no particular order.

- Siege Towers is restricted to that of the English language, as all documentation and game display text will be shown in English.

- Siege Towers is programmed specifically for the Android OS platform 2.2. It will not guarantee to be functional on any other platform or device running other operating systems.

- Siege Towers is its own separate application that is not an extension of another 3rd party application.

- Siege Towers will be deployed as a final product as of now. There will be no maintenance (updates, revisions) done after the completion of the game.

- Siege Towers will have a goal of 20 frames per second to keep consistent, stable game play for the user. Any frame rate greater than 20 will be optimal.

- Siege Towers will be developed within the mindset that it will be implemented on a mobile device. Resources such as RAM and memory space will be taken into consideration in terms of source code and assets (images, layouts etc.) for the game. Performance will vary depending on the model of the user's phone. The game will be implemented with the intent of making the game available to most Android users.

- Siege Towers uses a single virtual Linux server as the central messaging apparatus. Data transfers over the network are efficient as possible (serialized objects and 32-bit integers), so the server can support a large user base. The apache server and linux operating system has a much higher theoretical limit, so

the chief limitation is bandwidth. Beyond a certain number of concurrent requests for information, the internet connection of the server would become saturated. At this point, a distributed server architecture would be needed. Rackspace (the virtual hosting provider) guarantees a sustained throughput of 10Mbit / sec. After accounting for the networking overhead of the various protocols involved (HTTP, TCIP, IP, and Layer 2 protocols) it is conservatively esitmated that the server can handle hundreds of concurrent users. Beyond network bandwidth issues, the next bottleneck would be the processing power dedicated to the MySql server; this could be overcome by clustering or getting a large dedicated server.

6.0 Testing Issues

Each class of the game will be tested as they are made to ensure that they meet their specifications. The classes will then be integrated to create the overall program, and system testing will be implemented to ensure correctness and playability.

6.1 Classes of tests

6.1.1 General system testing

Black box testing will be used on each component and class of the program. Then an integrated black box system test will be implemented on the complete application.

6.1.2 Black Box Testing

Black box testing (no knowledge of the internal specification) testing will be done by playing the game. The user will not know the specifications of the game. Playing the game will then determine what features (i.e. Tower building timer length) need to be adjusted.

6.1.3 White Box Testing

White box (full or partial knowledge of the internal specification) testing will be conducted throughout the process of development. Each class will be tested individually to target specific mechanisms to ensure that each works as intended.

6.1.4 Screen Testing

Go through the sequence of creating a game, building a tower, choosing weapons, and playing through a game. The game should be paused during the building of the tower to ensure that it flows smoothly and doesn't continue to count down the timer while being paused. During the game play, shoot a projectile higher than the screen to make sure this is handled correctly.

6.1.5 Item Rendering

If items are rendered incorrectly it will be obvious immediately. Therefore, black-box testing will implicitly be carried out through the course of development. White-box testing can be carried out by trying to create a tower with a large number of blocks (larger than would be allowed in the game) in order to determine if any memory or performance issues are present.

6.1.6 Collision Detection

It will be obvious if weapons do not collide correctly with the towers, and will serve as useful testing in the black-box method. White-box testing can be carried out by creating abnormal towers to ensure that they obey in-game physics. Also, sprite sizes for both blocks and weapons should be tweaked to ensure that they are proportional to each other, and various sizes of each should be tested to determine the best flow for the game.

6.2 Expected software response

6.2.1 Screen transitions

Verify that the screen animations are smooth. When paused in the tower building phase, the timer should be stopped, and the screen should be dimmed to indicate that it is not in play. When un-paused, the screen should be undimmed and the screen should resume its previous state. During game play, the weapons should move smoothly, and there should not be lag when aiming the weapon. The weapon should be fired upon lifting the finger from the screen, and the animation of the collision should happen in real time.

6.2.2 Item Rendering

There should not be lag in the tower building phase. Many blocks should be able to be on the screen at once without causing performance issues. When changing the color (both in tower building and in gameplay) of an item, there should not be any sluggishness.

6.2.3 Collision Detection

Weapons should not be able to pass through impassible objects (the blocks in a tower).

6.3 Performance bounds

6.3.1 Screen transitions

Screen transitions should occur smoothly and with reasonable delay.

6.3.2 Item Rendering

Additional items added to the screen should not slow the response time of the game down.

6.3.3 Collision detection

Collisions with the tower should be smooth.

6.4 Identification of critical components

Components that require special attention to ensure proper functionality of the game are: GameManager, TowerBlock, BlockGeneratingSprite, and Game.

7.0 Appendices

7.1 Requirements traceability matrix

	Menu Screen	New Game Screen	Build Screen	Weapon Select Screen	Game Play Screen	Game Server
C2DM Handler						X
GameManager	X					X
NetworkManager	X	X				X
Game	X	X	X	X	X	
Enumerations			X	X	X	
TowerBlock		X	X		X	
GamePlayActivity					X	X
Weapon				X	X	
BlockGenerating Sprite			X	X	X	
WeaponsOverlayActivity			X			
NewGameActivity	X	X				X
TowerBuildingActivity			X			
BuildScreenOverlay			X			
HomeScreen	X					X

ResolutionManager	X		X			

7.2 Packaging and installation issues

SiegeTowers is packaged as an APK file. All android applications are all installed as APK files.