Unrein, Giang, Yang, Berg

DESIGN MANUAL

Hack The Planet
Peter Unrein, Hung Giang, Tongyu Yang, Adrian Berg
Dept. of Computer Science
Bucknell University
Lewisburg, PA 17837

INTRODUCTION

Battle city is a clone of a popular arcade game released by Namco in 1985. The game is programmed in a pattern inspired by an MVC design and other design mechanics explored over the course of CSCI 205. The program is run from a main file called "GameMain.java". The purpose of this code is to instantiate the view, the menu and the board. The view is a standard view class developed through the Netbeans swing gui builder. The menu class (Menu.java) is primarily a JPanel, Action Listener and Key listener. The menu allows the user to start the game. Once the game is started the menu is taken out of the view and the board is put in place. The board is a class that oversees the running of the actual gameplay. The board class oversees the interactions between sub classes such as the player tank, bullets, the blocks and the AI tanks. The collision detection and sound are handled in utility files (CollisionUtility.java and SoundUtility.java). Utility classes are used for computation that could be isolated to a specific purpose but not a specific class. Other classes specific to in game objects are all under the umbrella of our Sprite class (Sprite.java). These classes include things like Blocks, Tanks, Bullets and TankAI. The Sprite Class defines basic necessities for any object that will be drawn.

1 USER STORIES

Our user stories were decided upon early and displayed in the chart below. The format of our user stories was As a/an [class] I want to [goal] so that I [reason] followed by an hour estimate. All of the user stories listed below were achieved by our current build. We have established some stretch goals that were outlined in our presentation.

Table 1: User Stories

As a/an	I want to...	so that...	Hours Estimate
Player	play the game offline	I could play the game whenever I want to	4
	be able to navigate the tank easily	I could happily enjoy the game	1
	know if I win the game	I have a satisfaction of playing the game	6
	know what is my score	I could improve it next time	1
	listen to different sounds while playing the game	I would not be bored	4
Tank	be able to move on the map	I could move to destroy enemies	4
	be able to shoot bullets	I could perform my actions	1.5
	be able to destroy walls	I could move more easily	2
	be able to destroy other tanks	I could kill enemies/fulfill my duties	3
	be able to shoot the base	I could win/lose	1

	have multiple lives	I could stay in the game longer and will be harder to be destroyed	0.5
	be able to power up	I could have better performance	1
Enemy	create some difficulties to the player	the game will be more interesting	4
	shoot the base	I could win	0.5
	move at a certain speed	I could destroy more and avoid bullets from the player	0.5
	shoot at a certain speed	I could destroy more	0.5
Bullet	be destructive	I can destory obstacles and tanks	0.5
	be able to be shoot by tanks	I could become the weapon of the tanks	1
	move in one direction	I could obey the real life rules	1
	have different speed at different situations	I could be more useful to the tanks	1
Base	be shown on the map	I could represent the win/loss of the player	0.5
	change to a white flag when I am destroyed	the player would know win/lose	0.25
Tree	be shown on the map	I can obfuscate bullet and tanks to make the game interesting	0.25
Steel	block bullet and tanks	I can make the game interesting	0.25
Water	block all the tanks and only let bullet pass through	I can make the game interesting	0.25
Brick	be able to be detroyed by the tanks	I could make the game challenging and protect the player's tank	0.25
PowerUp	be picked up by the player's tank	I could power up the player's tank	1.5
	improve different properties of the play's tank	I could make the game more interesting and be more useful to the players	3
	randomly show up on the screen	I could pick up by the player	1
Map	be generated/drawn	I could provide a space for the game to take place	1.5
	be updated at different stages	the game could be interesting	3
Game	run game logic	the game could run properly	3
	detect collision	the walls/bullets and tanks could be properly restricted	2
Edge/Side bar	show the extra lives of the player	the player could play wisely	2
	show the remaining enemies of the stage	the player could play wisely	1
	show the current stage of the game	the player could know which stage it is	1
Menu	be a guide to the player	the player could enter the game easily	1.5
ScoreBoard	keep track of the score of the player	the player could improve their performance each time	2.5

While these goals seem like a long list many are achieved quite easily. The Player user stories were solved in separate parts of the code and are more general. The game in is only playable offline, the player starts the game through a menu and exits the game through a game over menu. The game is playable and has easy to use controls to move the tank and shoot (arrow keys to move, space to shoot). The Enemy tanks come in

four varieties and have a smart enough difficulty to provide a challenge to an average player (one developer has never beaten level 3). The side bar accurately displays the players remaining lives and current stage accurately. We have all of the block types we wanted working properly and a scoreboard that appears once the player loses. The sound effects from the original game are properly triggered by the correct events in the game play.

2 OBJECT ORIENTED DESIGN

In total we used thirty-two classes to create our game. While this is a lot of classes most of them are sub classes of our Sprite class. The three most essential classes in our code are the Board, Sprite and Collision Utility classes. Included in figure 1 (located in appendix) is a map of our UML with these classes highlighted. The Board class is what updates each object in the game. Object in the game are all sprites and they are all kept in arraylists in the Board class. The Board class has an arraylist for enemy tanks, blocks (stationary sprites that are part of the map), powerups, animations and bullets. The Collision Utility supports the board class by calculating collisions between objects on these lists and the players tank. All of the objects in these arrays are Sprites and because they are all sprites they are all drawn the same way by the board (albeit in groups for the sake of priority and clarity).

2.1 The Board Class

Board	
Responsibilities	**Collaborators**
Keep track of blocks, bullets, and tanks Moves to next stage Checks for game over	Blocks Bullets Tank TankAI CollisionUtility BoardUtility PowerUp Animation

Our board class was the largest class in our game and key to the integration of all the other classes. Above is the Board classes CRC and in the appendix figure 2 shows the UML diagram. The key functions in the Board class are the draw functions and the update functions. The draw functions (such as drawObjects) draw the arrays of objects and parts of the component and are all called by the update functions. The update functions are called on each tick of our timer and update the logic of each object in the game (bullet, enemy, player etc.). By quickly alternating between these two and calling collision utility as well in very quick succession our game runs very smoothly and the board stays accurately updated.

The board is also responsible for loading levels, transitioning to the next level when the current level is beaten and showing the game over screen when the player loses. The board checks for stage completes as well as losses by checking three conditions. Any stage will end on three conditions. If they player runs out of lives or the base is destroyed, there will be a game over. A game over is signified by a stoppage of gameplay and red “game over” text appearing on screen. If the player kills all the tanks in the level then the player will complete the stage and the next one will load. The figure below is a code snippet of how our board loads the next level.

```
/**
 * Call initBoard to enter next Level when no enemy in the list.
 */
public void nextLevel() {
    if (enemy.isEmpty()) {
        stage += 1;
        numAI = 0;
        numEnemies = goal;
        clearBoard();

        initBlocks();
        CollisionUtility.loadCollisionUtility(blocks, animations);
        BoardUtility.loadBoardUtility(enemy, blocks, animations, powerUps,
                                      tank);

    }
}
```

Figure 2.1.1: between stages the board is cleared and a new set of blocks are initialized

We load 35 levels in total from an online database. The maps online are 2-D arrays that are encoded by symbols representing blocks. We take these arrays and decode/encode them into arraylists of our own enum called Blocktype.

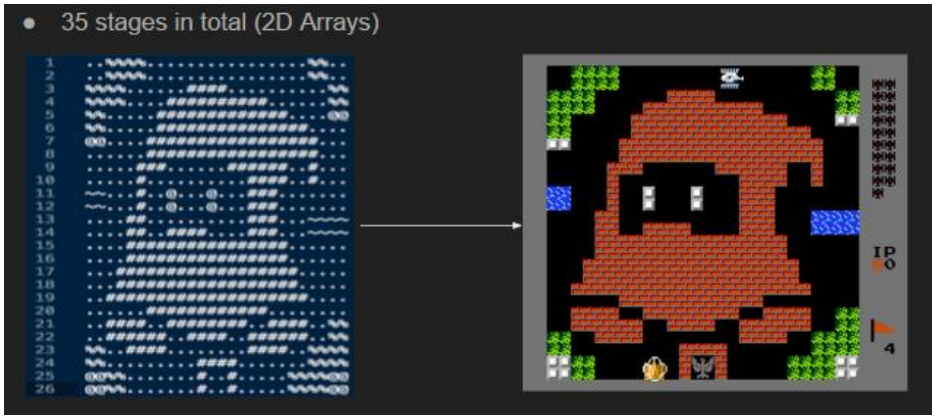


Figure 2.1.2: An encoded array found online and the final result

The "edge" which is the border around the game area is drawn within the board class and reports enemy count, life count and stage number on the right hand side.

2.2 The Sprite Class

The Sprite Class sets definitions for any object that is going to be drawn by the board

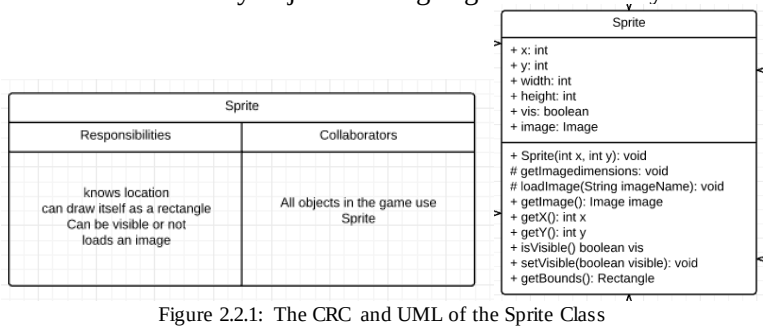


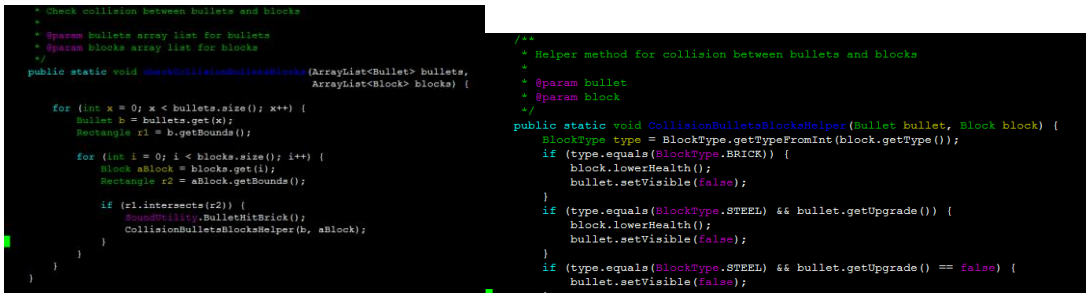
Figure 2.2.1: The CRC and UML of the Sprite Class

This class definition was key to keeping the many types of object in our game normalized. By having a Sprite class we're able to draw all objects the same. Considering we have to draw blocks which come in types, bullets, a base, powerups and enemies this is a very smart design choice. Moving objects like the Tanks and Bullets have move functions that are called to change their position. Animations like the shield effect and the explosions are under a subclass of Sprite called Animation. All the block types as well as

the edge of the board are organized as subclasses under a Sprite subclass called Block. By organizing our classes this way we were able to keep a lot of our very specific game logic under one umbrella while still tailoring it to each specific objects needs.

2.3 The Collision Utility

The Collision Utility is used by the board to scan it's arraylists of objects against each other and see what kind of interactions are happening. The objects that can interact with each other are the blocks, bullets and tanks. The bullets can hit the blocks and tanks but not each other, the enemy AI cannot collide with each other (they would not be smart enough to avoid bumping into each other all the time). Below is a snippet of our collision detection algorithm.



```
    * Check collision between bullets and blocks
    *
    * @param bullets array list for bullets
    * @param blocks array list for blocks
    */
   public static void checkCollisionBulletsBlocks(ArrayList<Bullet> bullets,
                                                  ArrayList<Block> blocks) {

       for (int x = 0; x < bullets.size(); x++) {
           Bullet b = bullets.get(x);
           Rectangle r1 = b.getBounds();

           for (int i = 0; i < blocks.size(); i++) {
               Block aBlock = blocks.get(i);
               Rectangle r2 = aBlock.getBounds();

               if (r1.intersects(r2)) {
                   SoundUtility.BulletHitBrick();
                   CollisionBulletsBlocksHelper(b, aBlock);
               }
           }
       }
   }
```

```
    /**
     * Helper method for collision between bullets and blocks
     *
     * @param bullet
     * @param block
     */
    public static void CollisionBulletsBlocksHelper(Bullet bullet, Block block) {
        BlockType type = BlockType.getTypeFromInt(block.getType());
        if (type.equals(BlockType.BRICK)) {
            block.lowerHealth();
            bullet.setVisible(false);
        }
        if (type.equals(BlockType.STEEL) && bullet.getUpgrade()) {
            block.lowerHealth();
            bullet.setVisible(false);
        }
        if (type.equals(BlockType.STEEL) && bullet.getUpgrade() == false) {
            bullet.setVisible(false);
```

Figure 2.3.1: The collision detection as well as the helper that handle the situation

Note that in figure 2.3.1 there is a loop to detect the collision and a helper method to handle those collisions. Because the helper is called in the midst of the loop it does not remove any object but sets it to invisible. This is so the object disappears as quickly as possible but is not removed so that collisions with multiple objects can happen simultaneously. When the Board redraws items it removes the invisible ones. We implemented this method of collision very purposefully and it makes a lot of sense to us. By keeping the objects in the arraylist during collision detection after they've collided, you avoid errors where a method will detect one object colliding twice and therefore try to remove it twice only to come into an error when trying to remove that object a second time.

Separating the collision code into checks and helpers also helps to maintain design clarity. Every check algorithm will be written essentially the same, the only difference is which arraylists are being checked against each other (we could even check the same arraylist against itself). The helper methods however each had to be specialized to handle the special situation of that collision type. For example a bullet colliding with a block leads to a different outcome depending on what block type the block is or how upgraded the bullet is.

APPENDIX

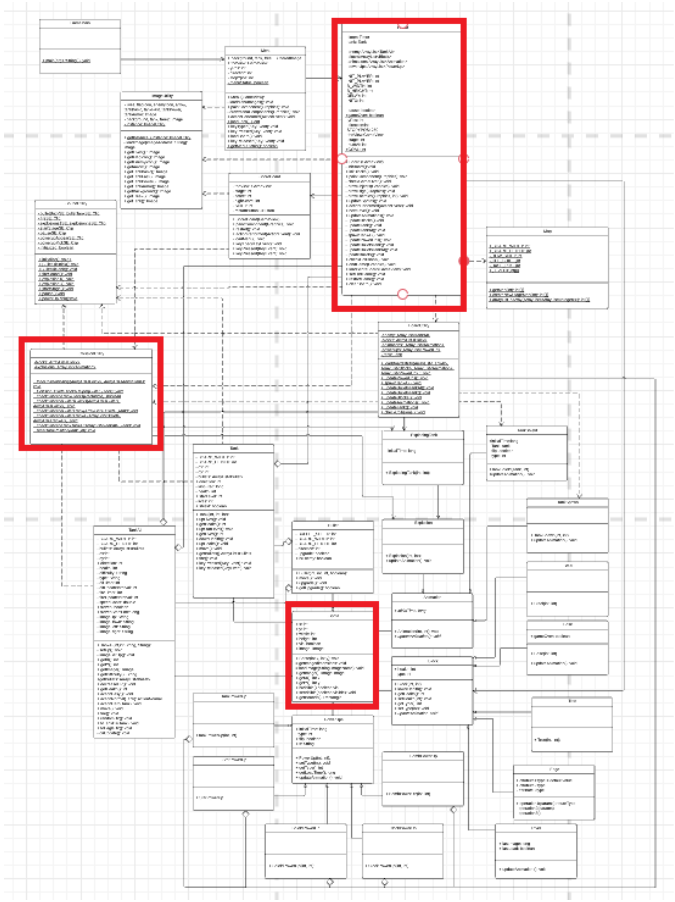


Figure A 1: Board(top right), CollisionUtility (Center left) and Sprit (Bottom Center) are our three most important classes

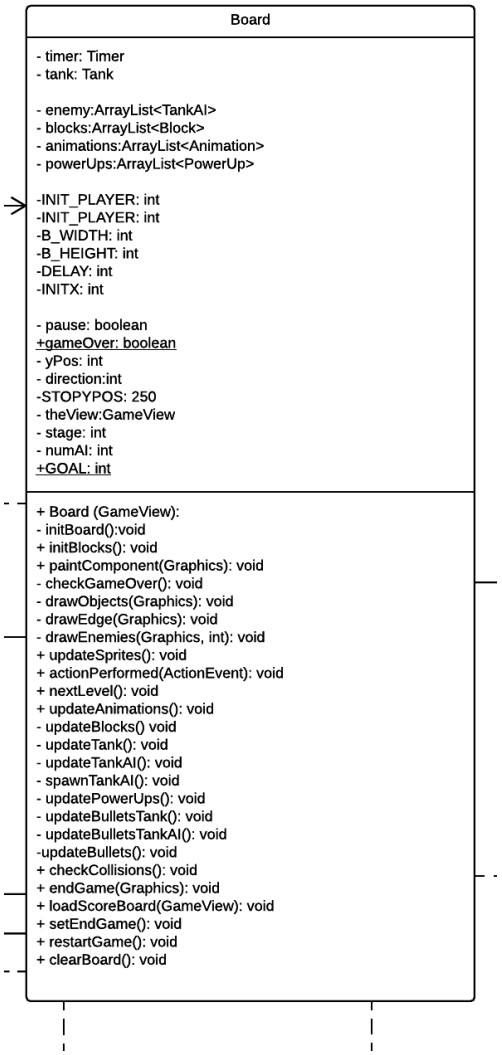


Figure A 2: Board UML