# <^> TRIFORCE <^>

## Combined Documentation

by Gabe Pike

CMPS 335

Spring 2012

Team 10

Other Team Member: Brandon Hinesley

# Table of Contents

# Section 1.1: Requirements Workflow

## 1.1.1 Initial understanding of the project

### General

Triforce is a remake of the classic SNES game, Tetris Attack. It is called Triforce in part because of the core game mechanic where one must align at least three matching blocks. In the game, a player controls a cursor that lets him swap two adjacent blocks horizontally. Combinations of 3 or more blocks in a row will break the blocks, causing the blocks above them to fall. The game mechanics allow the player to use their skills and reaction time to gain great rewards. The game mechanics are flexible and allow for several fun modes of play. Figure 1 shows a screenshot of gameplay in Tetris Attack.


*Figure 1: This screenshot shows a game with two players. Each player controls the white cursor on their grid of blocks. A player presses a button two swap the blocks inside the cursor.*

**Menu**

The user experience should begin with opening the game to the main menu. The menu will contain buttons for Play, Settings, Help, Credits/About, and Quit. The Settings button will open a new menu to configure settings such as themes, sounds, and difficulty. Help will tell the user how to play. When the user selects Play, they will be taken to another menu to select what mode they want to play. When they select the mode, they will be taken to the actual game. It will start after a timer of three seconds. The gameplay area will consist of a grid of blocks and a cursor for the player to control, as seen in Figure 1. The player uses the keyboard or mouse to move the cursor to an adjacent block horizontally or vertically. The player must break blocks by aligning three or more of the same color vertically or horizontally.

**Controls/Input**

The controls to Triforce are rather simple. The user only needs to concern himself with the following input (input that will use the same binding is put in the same bullet point):

- Navigate the menus; Move the cursor in four directions
- Select a menu item; Swap blocks
- Boost the speed of play
- Pause/Unpause; Exit

We want to support multiple methods of input. It will be possible to use a keyboard, mouse, or Xbox 360 controller to control everything in the game. This will be done by making generic bindings that are not specific to any kind of input device. Then, we will map each input method from each input device to the generic bindings.

**Gameplay and Mechanics**

Triforce has potential for several modes of play. Due to time constraints, we will not likely be able to implement them all. At a minimum, we want to implement Endurance Mode, where a player must survive as long as he can while blocks are pushed onto the play area at accelerating speeds. If time permits, we will also implement Versus Mode. In Versus Mode, two players build up large combos and chains, and drop them on each other. It could be played on the input device or over a network connection. Due to time constraints and experience, it is unlikely we will implement any sort of artifical intelligence.

The game play of Triforce looks rather simple to a player, but its inner workings are actually rather complex. The main game mechanics are combos and chains. The mechanics of combos are the easiest to explain. A combo is achieved by aligning at least three matching blocks together. However, you can combine sets of blocks by positioning things right before moving them. Below are some examples of possible combos.

*Figure 2: 3x Combo*

*Figure 3: 4x Combo*

*Figure 4: 6x Combo (2 sets of blocks)*

Chains are more difficult to pull off. Simple chains require a bit of planning, but it is possible to pull off some very complex, well-thought-out chains. A chain is when at least one combo causes blocks to fall directly into another combo. The figures below feature some examples of chains in ascending difficulty.

*Figure 5: Basic Chain*

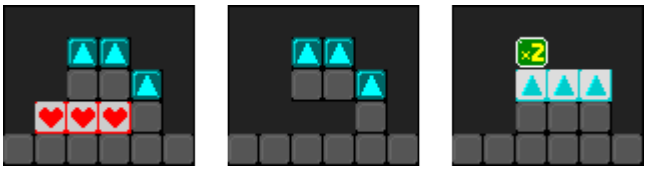



*Figure 6: Raised Skill Chain*

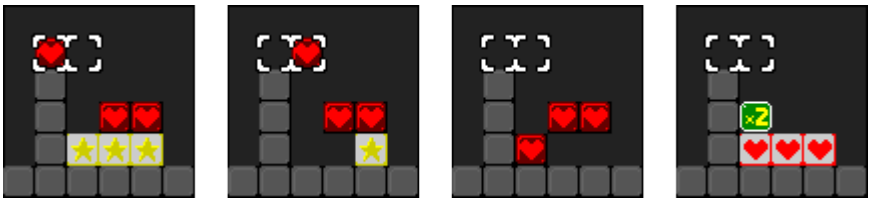



*Figure 7: Advanced Chain*

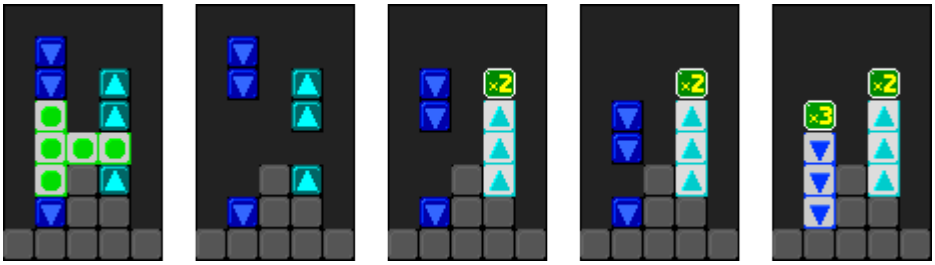



*Figure 8: Time-Lag Skill Chain*

Tools, Libraries, and Version Control:

We will be using Visual Studio 2010 as our development environment during the production Triforce. The game will require the GLUT library for graphics and sounds, and the 2DGraphics library. Due to some limitations of the 2DGraphics library, we will be adding several features to the header file. It will run in a Windows environment and compiled an x86 architecture. We are going to manage our source code and resources with Git and use Github to host our Git repository. We chose Git because both of us know the benefits of version control, and I have experience with Git. Github was chosen because of its many rich features for developers, such its bug/enhancement ticket system, commit tracking tools, wiki, and online code viewing.

## Glossary

Combo: A game mechanic where a player aligns at least three matching blocks

Chain: A game mechanic that requires a player to use smart timing and positining to cause combos to generate more combos from falling blocks.

OpenGL: An open-source, cross-platform API for creating 2D and 3D computer graphics.

GLUT (OpenGL Utility Toolkit): A closed-source procedural library written in C that contains utilities for OpenGL programs and input devices.

2DGraphics Library: A closed-source interface to GLUT that contains library functions useful for games, created by Dr. Arif Wani.

Version Control: Management of changes to documents, computer programs, or other collections of information.

Repository: A  data structure, usually stored on a server, that contains a set of files and directories, historical record of changes in the repository, set of commit objects, and a set of references to commit objects, called heads.

Git: A version control system that is fast, local, and distributed.

Github: A popular website for hosting Git repositories that is free to use.

Commit: A single revision in a revision control system.

Visual Studio 2010: An integrated development environment for Windows.

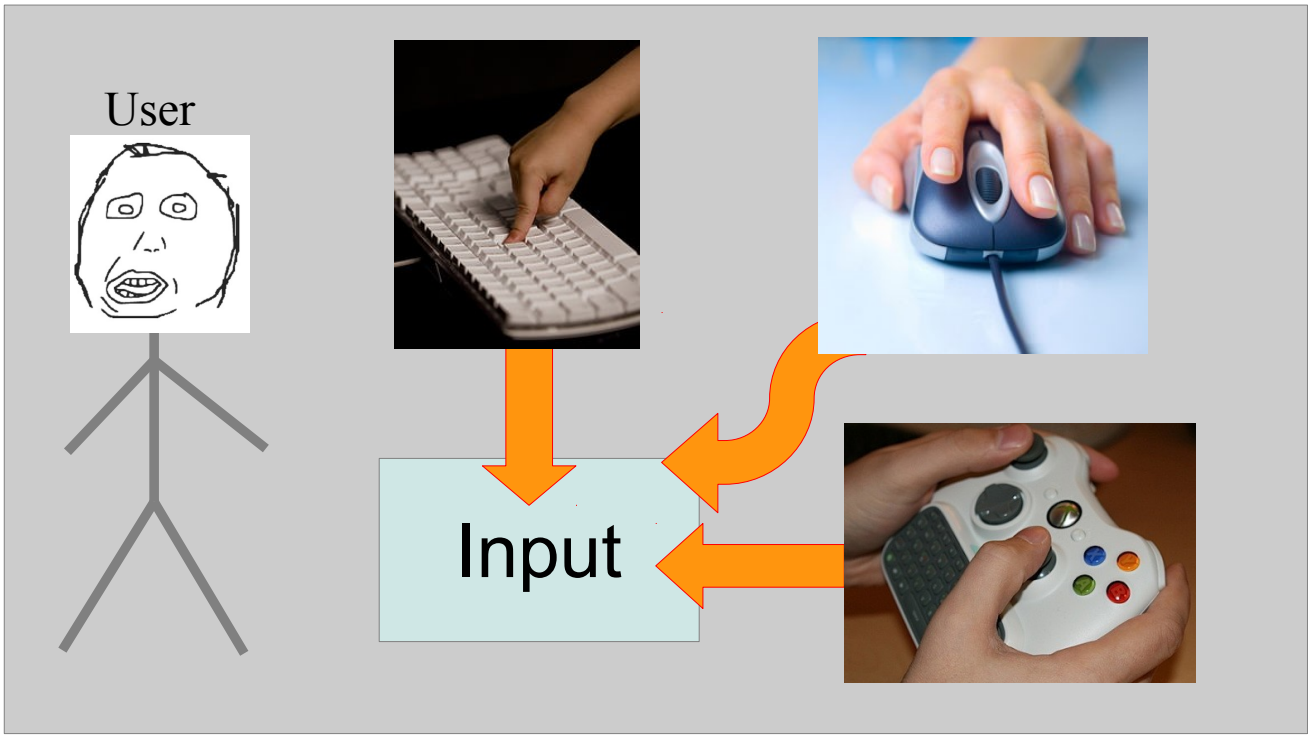
### 1.1.2 Initial Business Model

When a user runs the game, the first thing that happens is that the configuration file is loaded from disk. The configuration file contains key bindings for input, the preferred default theme, and theme directory location. After loading the configuration file, the initial menu is displayed to the user. The initial menu will display a background image and the buttons used to transition to other states of the game. The menu contains the following buttons: Play, Settings, Help, About, and Quit.

Use-Case 1: User runs the executable


Brief Description: When the user runs the programs, a configuration file is loaded and the main menu is displayed.

Use-Case 2: User activates an input event



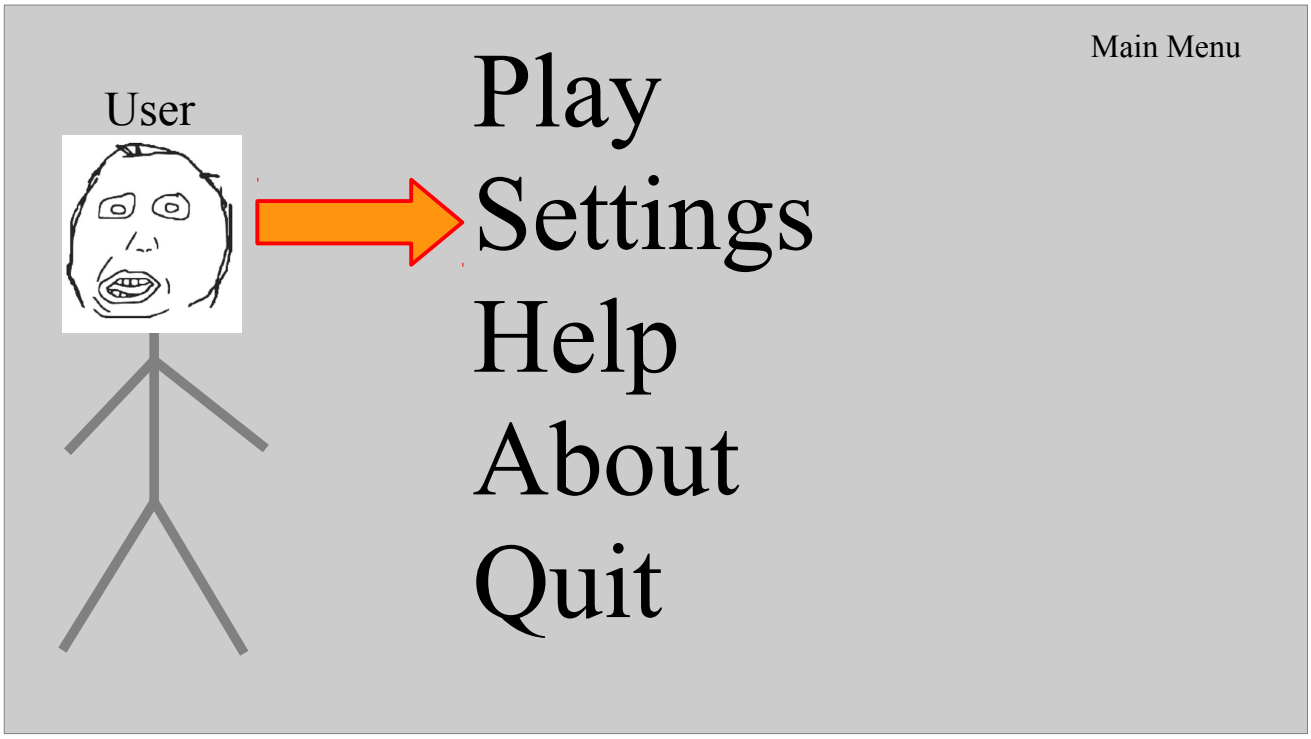

Brief Description: When a user activates a binding on an input device, it is handed by a generic interface via mappings and callback routines.

Use-Case 3: User selects Play


Brief Description:

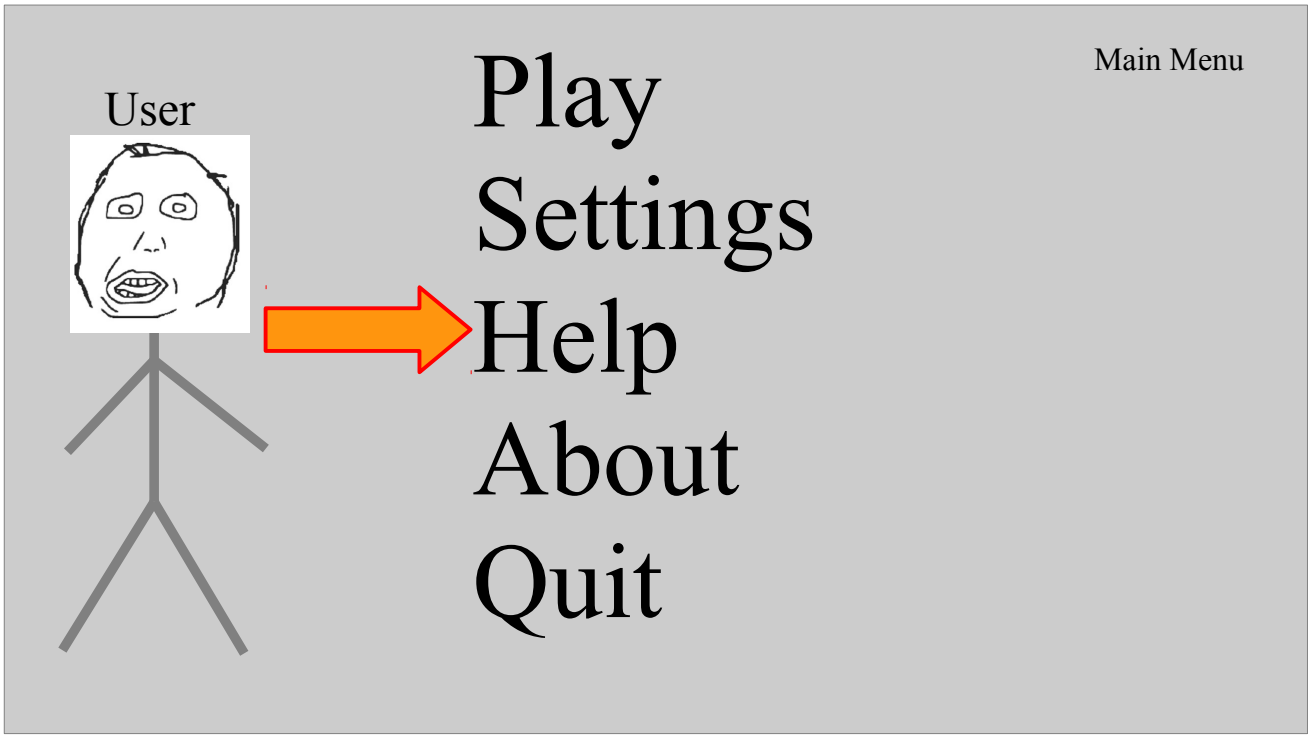
The user selects play on the main menu and the game begins.

Use-case 4: User selects Settings




Brief Description: The user selects Settings and the settings menu is displayed. On the settings menu, the user can change controls, audio, theme, and game play settings.

Use-case 5: User selects Help




Brief Description: The Help button will display a small tutorial about how to play.

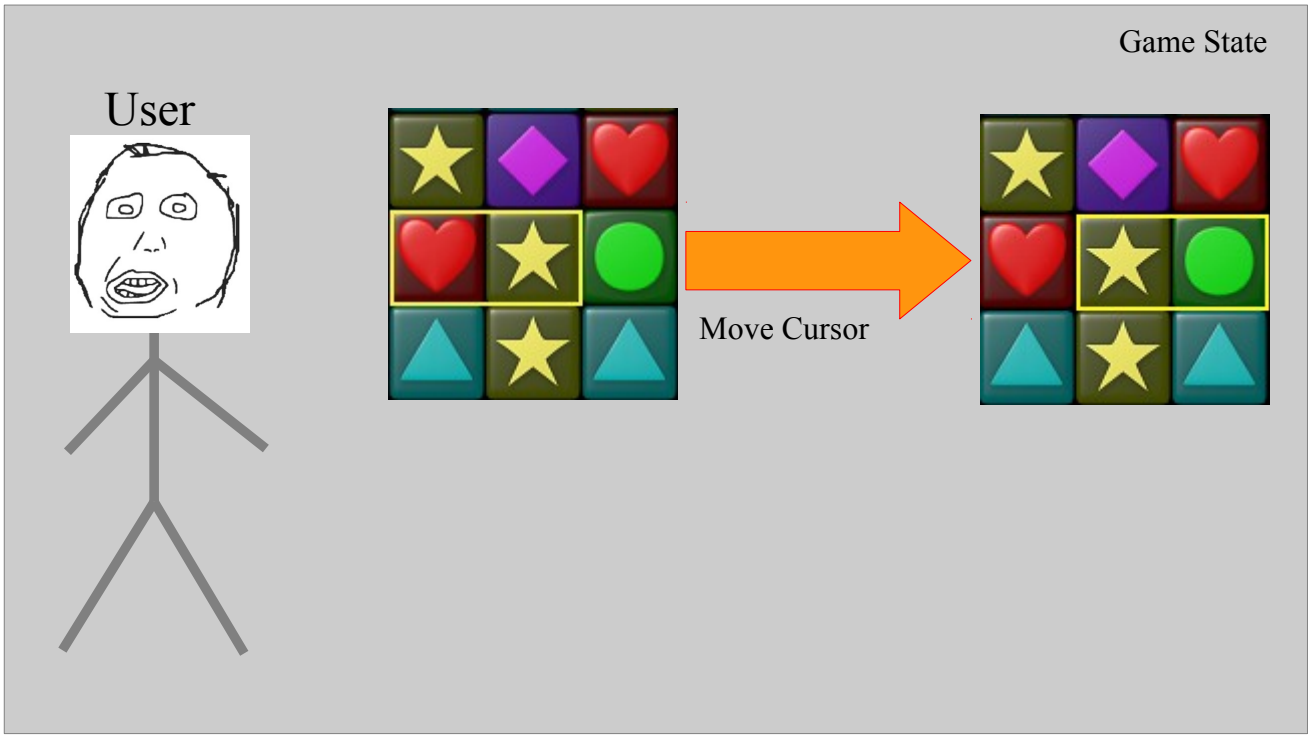
Use-case 6: User selects About


Brief Description: The About button will display information about the authors, licensing information, and a brief description of the game.

Use-Case 7: Use selects Quit

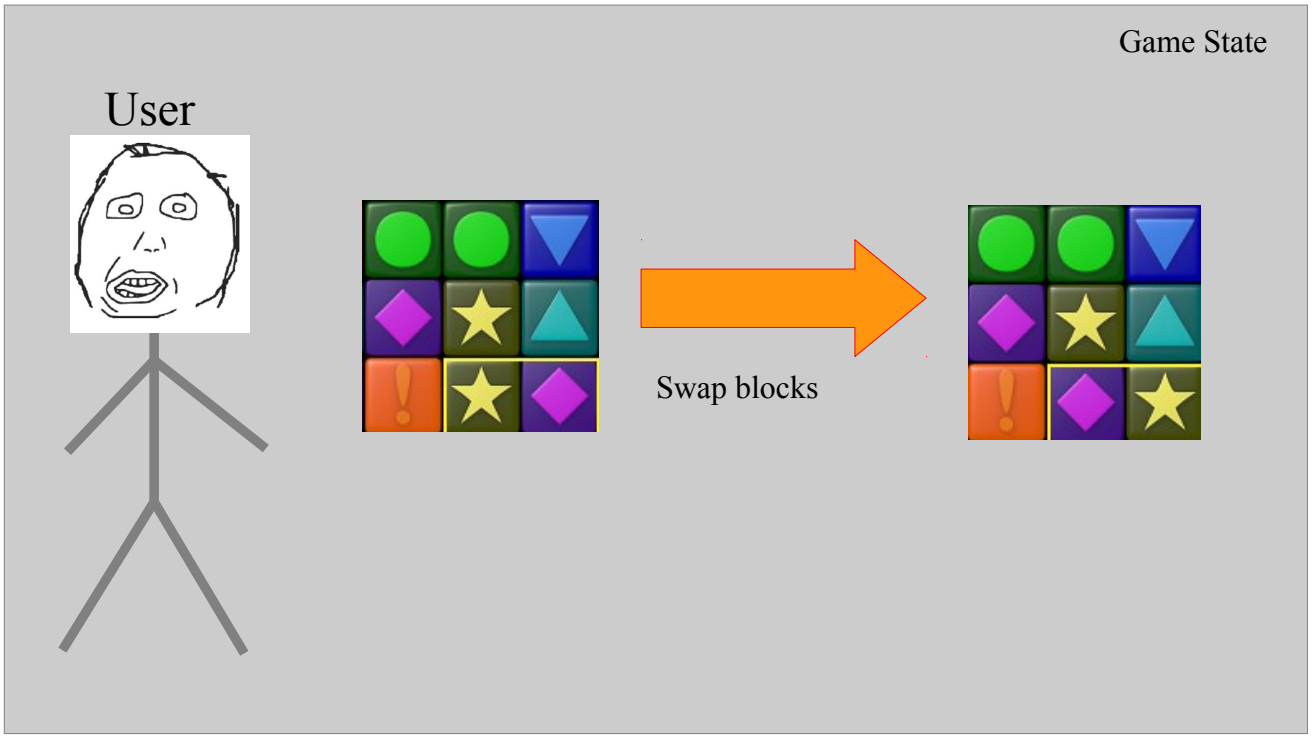

Brief Description: The game will exit when the user selects Quit.

User-Case 8: User moves the cursor



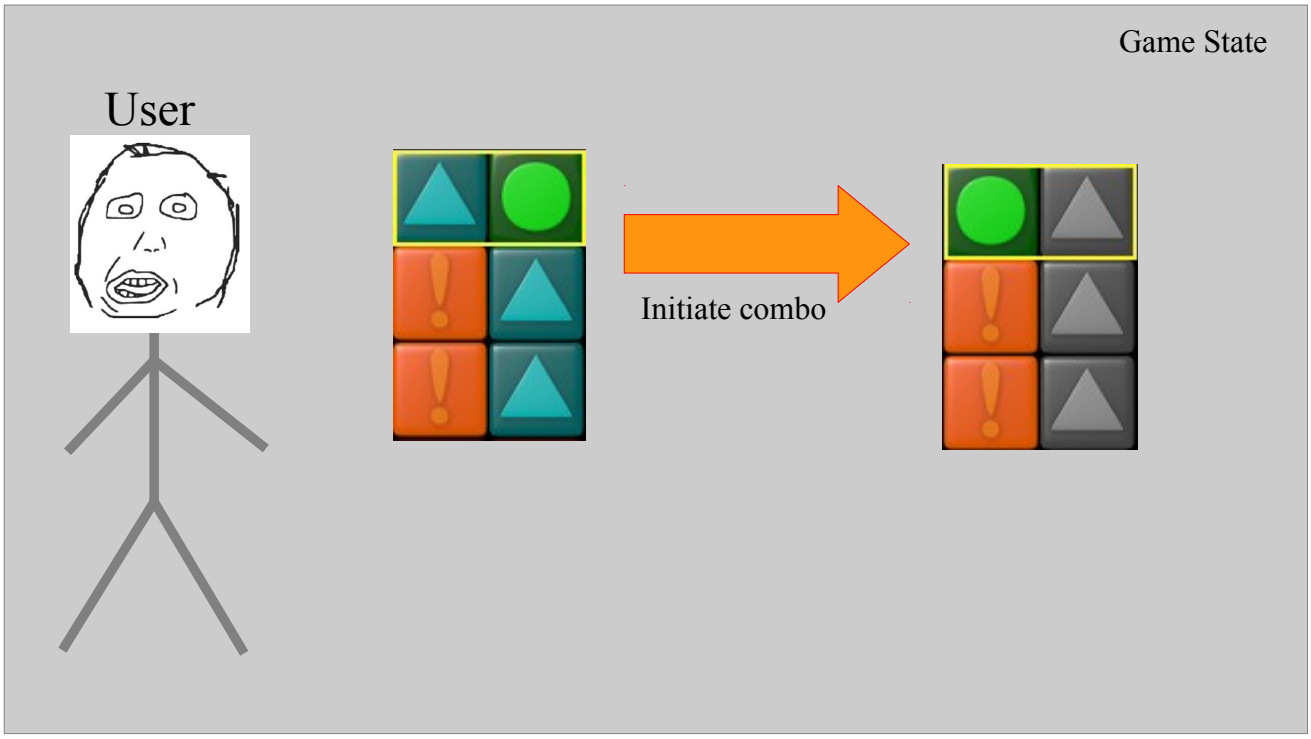

Brief Description: A user moves the cursor and its position is updated to a new cell.

User-Case 9: User swaps blocks




Brief Description: When the user swaps blocks, their positions in the grid are swapped and combo and fall detection must occur.

User-Case 10: User activates a combo




Brief Description: If a user activates a combo, the blocks will momentarily pause, then break, then cause all blocks above them (if any) to fall.

### 1.1.3 Initial Requirements

### 1.1.4 Detailed Requirements

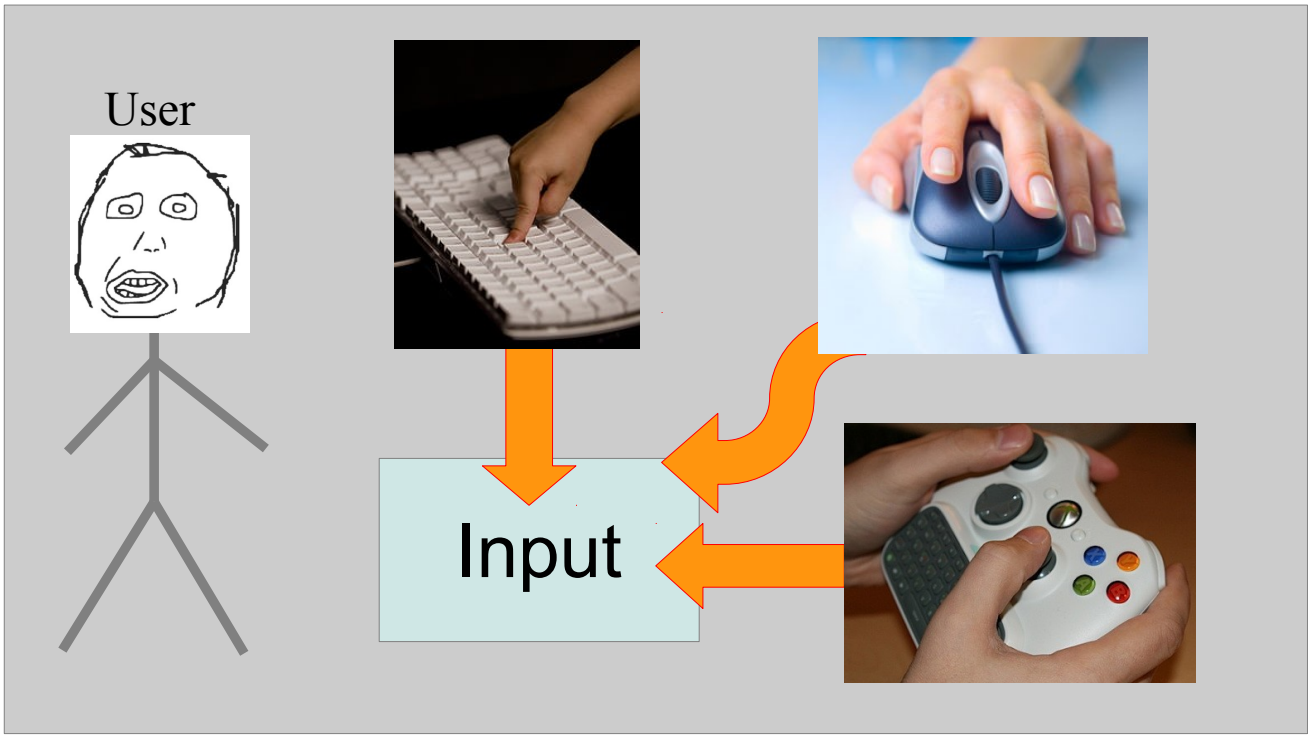
### 1.1.5 Iterated case-use diagrams

Use-Case 1: User runs the executable


Step-By-Step Description:

1. Run the executable program
2. Configuration file is loaded. (load sane defaults if no config file)
3. initialize menu objects
4. load resources for main menu
5. Change game state to Menu
6. Current menu is set to MainMenu
7. Display the menu buttons
8. Accept Input
9. Handle Input

Use-Case 2: User activates an input event




Step-By-Step Description:

When a user activates an input event, such as from a keyboard, mouse, or Xbox 360 controller, it is handled by a single interface. This interface is a pseudo-device that maintains keybindings for the actual device. For instance, the 'A' key, 'A' button, and left-click can all map to a binding that handles selecting menu items and swapping blocks.

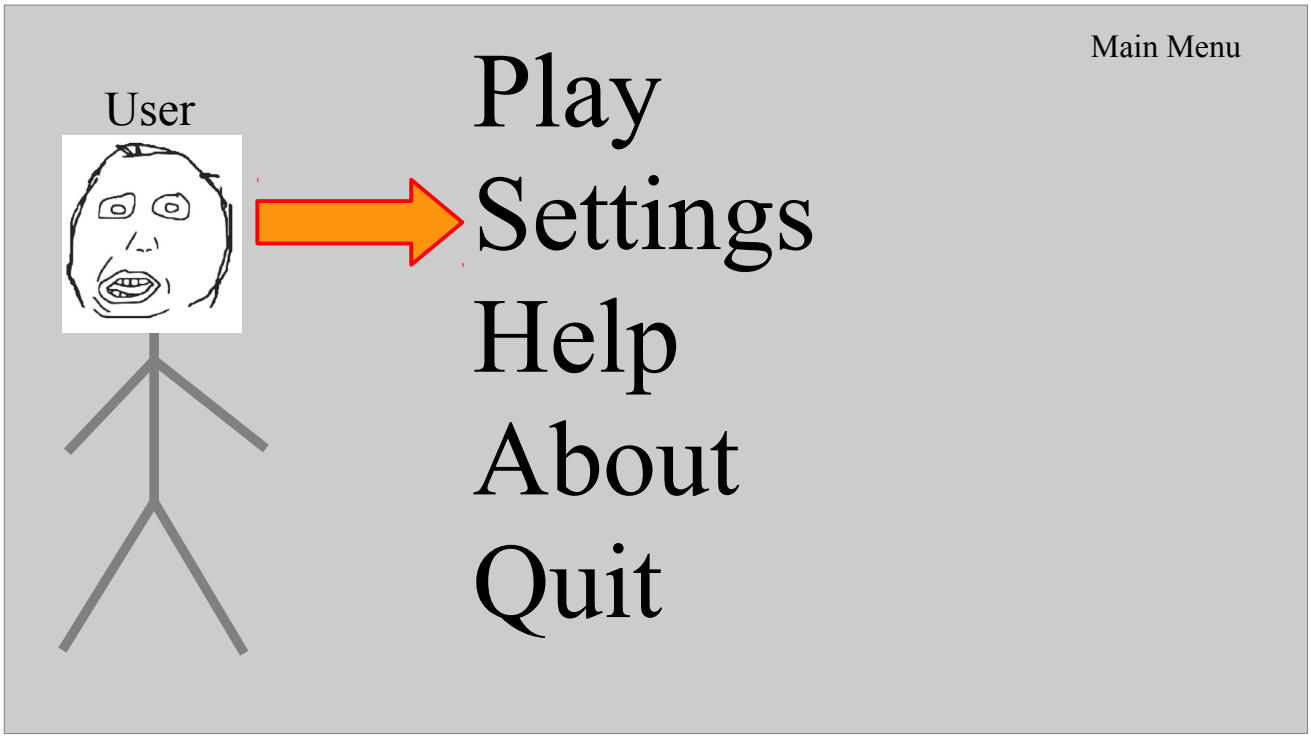
1. User activates a registered key binding
2. Current context passes it to the Input interface.
3. Input interface convert it to a generic key binding
4. Handle action via the registered callback function

Use-Case 3: User selects Play


Step-By-Step Description:

1. User selects Play
2. Input interface handles action via callback for Play
3. Callback function sets game state to Play
4. Game state is changed to Play
5. Initialized Game objects
6. Load resources for game objects
7. Begin a countdown
8. Start the game

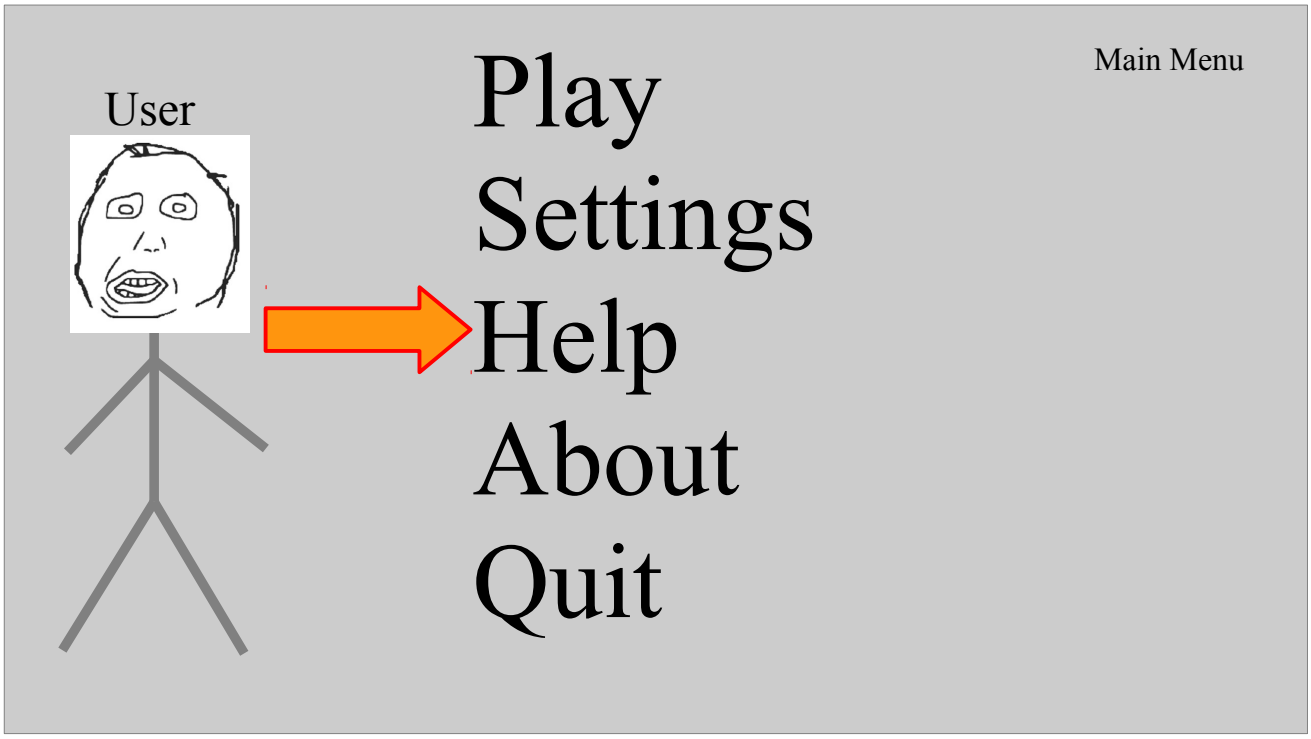
Use-case 4: User selects Settings




Step-By-Step Description:

1. User selects Settings
2. CurrentMenu is set to Settings
3. Settings menu objects are initialized
4. Settings menu resources are loaded
5. Control passed to the user
6. User clicks Save to save settings to configuration file
7. User clicks Back to go back to main menu

Use-case 5: User selects Help




Step-By-Step Description:

1. User selects Help
2. CurrentMenu is set to Help
3. Objects are initialized
4. Resources are loaded
5. Small tutorial is displayed, aided with text and graphics
6. User can click Back to go to main menu

Use-case 6: User selects About


Step-By-Step Description:

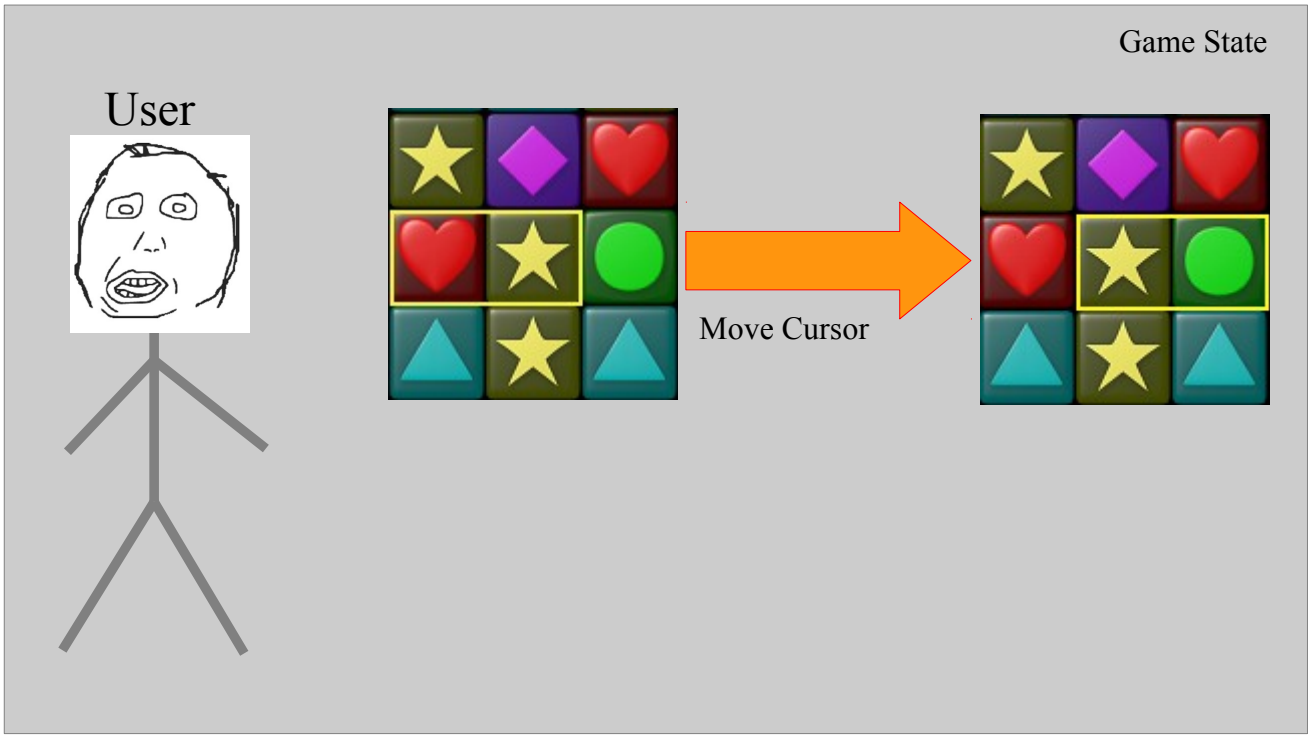
1. User selects About
2. CurrentMenu is set to About
3. Objects are initialized
4. Resources are loaded
5. Brief description of game, licensing/copywrite info, and credits are displayed
6. User can click Back to go to main menu

Use-Case 7: Use selects Quit


Step-By-Step Description:

1. User selects Quit
2. Prompt user, asking if they are sure (if time permits)
3. Free objects and resources
4. Exit the program

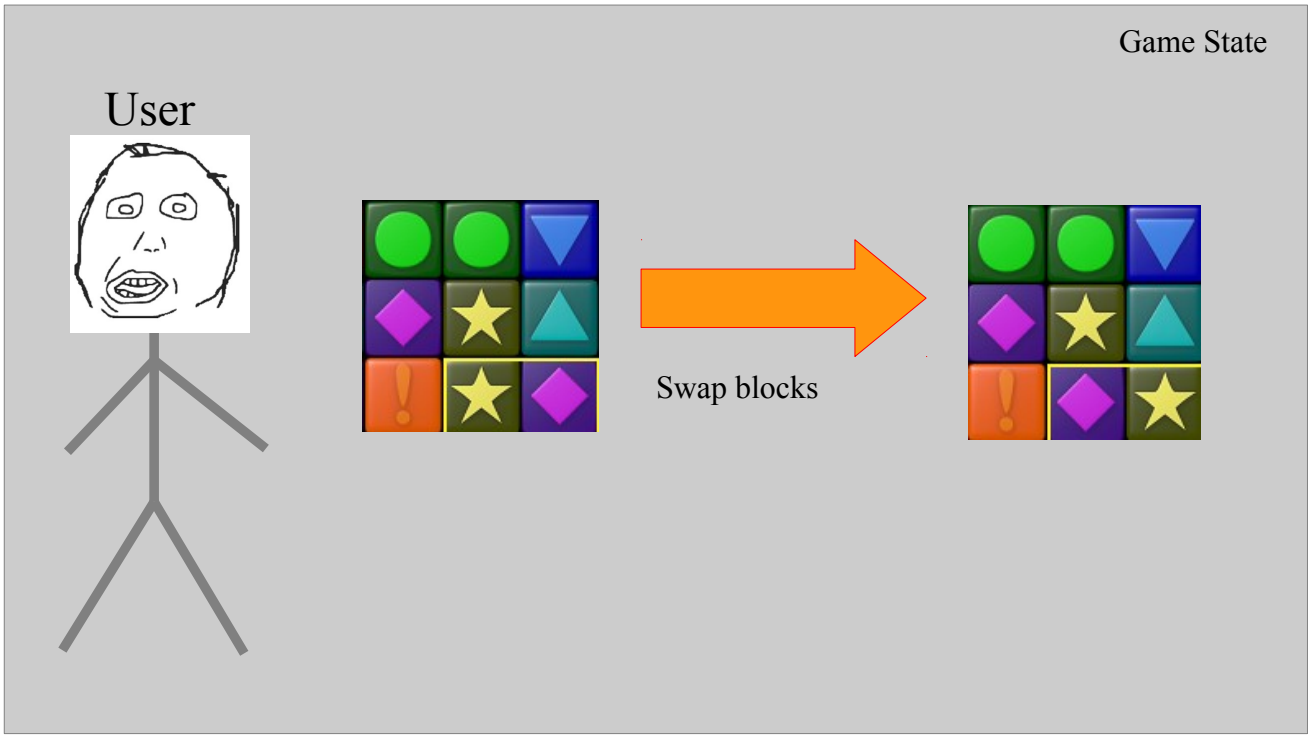
User-Case 8: User moves the cursor




Step-By-Step Description:

1. Check boundaries: Make sure cursor does not leave the game area
2. Calculate new position as an offset of the Grid's position, given the row, column, and block length

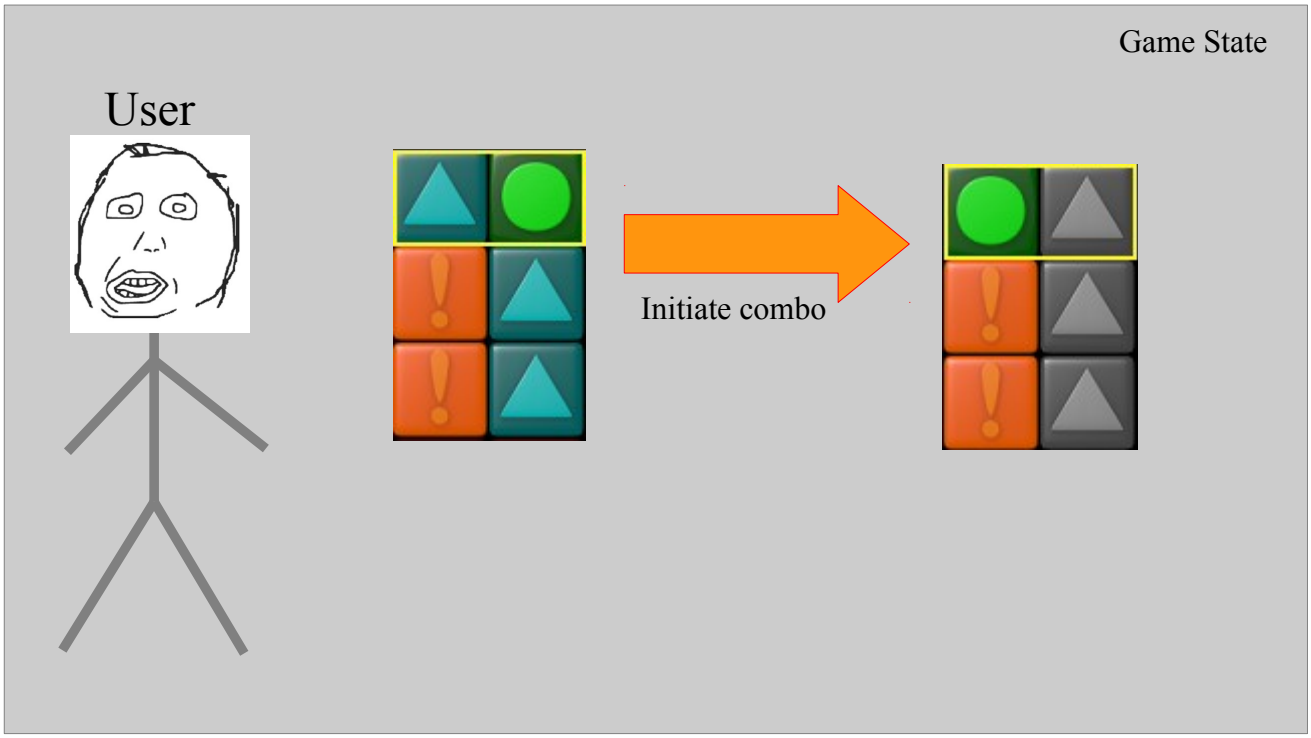
User-Case 9: User activates a combo




Step-By-Step Description:

1. Swap the positions of the blocks in the Grid data structure
2. Detect possible new combos
3. Detect possible new falling states

User-Case 10: User activates a combo




Step-By-Step Description:

1. Calculate the interval that the combo must last, based on the number of blocks broken
2. Set the Blocks being broken to a Combo state
3. Set the Grid to a Combo state
4. Break the blocks after the interval elapses
5. Set blocks above the broken combo (if any) to a Falling state
6. Detect Combos on newly fallen blocks

## 1.2 Walkthrough

-By Brandon Hinesley

- Spell check (all I saw was artifical -> artificial)
- Explain how blocks flow from bottom up, since this is not typical of block games.
- Need to cover garbage blocks.
- Do not need use-case for starting the game and some menu options
- Draw use-case diagrams correctly, according to software engineering standards

## 1.3 Revised Requirements after walkthrough

### 1.3.1 Initial understanding of the project

#### General

Triforce is a remake of the classic SNES game, Tetris Attack. It is called Triforce in part because of the core game mechanic where one must align at least three matching blocks. In the game, a player controls a cursor that lets him swap two adjacent blocks horizontally. Combinations of 3 or more blocks in a row will break the blocks, causing the blocks above them to fall. The game mechanics allow the player to use their skills and reaction time to gain great rewards. The game mechanics are flexible and allow for several fun modes of play. Figure 1 shows a screenshot of gameplay in Tetris Attack.


*Figure 9: This screenshot shows a game with two players. Each player controls the white cursor on their grid of blocks. A player presses a button two swap the blocks inside the cursor.*

**Menu**

The user experience should begin with opening the game to the main menu. The menu will contain buttons for Play, Settings, Help, Credits/About, and Quit. The Settings button will open a new menu to configure settings such as themes, sounds, and difficulty. Help will tell the user how to play. When the user selects Play, they will be taken to another menu to select what mode they want to play. When they select the mode, they will be taken to the actual game. It will start after a timer of three seconds. The gameplay area will consist of a grid of blocks and a cursor for the player to control, as seen in Figure 1. The player uses the keyboard or mouse to move the cursor to an adjacent block horizontally or vertically. The player must break blocks by aligning three or more of the same color vertically or horizontally.

**Controls/Input**

The controls to Triforce are rather simple. The user only needs to concern himself with the following input (input that will use the same binding is put in the same bullet point):

- Navigate the menus; Move the cursor in four directions
- Select a menu item; Swap blocks
- Boost the speed of play
- Pause/Unpause; Exit

We want to support multiple methods of input. It will be possible to use a keyboard, mouse, or Xbox 360 controller to control everything in the game. This will be done by making generic bindings that are not specific to any kind of input device. Then, we will map each input method from each input device to the generic bindings.

**Gameplay and Mechanics**

Triforce has potential for several modes of play. Due to time constraints, we will not likely be able to implement them all. At a minimum, we want to implement Endurance Mode, where a player must survive as long as he can as the game gets increasingly more difficult. If time permits, we will also implement Versus Mode. In Versus Mode, two players play against each other by building chains of combos to build garbage blocks that are dropped on the enemy (explained below). In theory, two players could play on one display with separate input (or sharing) input devices, or play over a network connection. Other modes of play allow a user to

play against enemy agents. However, due to time constraints and experience, it is unlikely we will implement any modes that employ artificial intelligence.

In the modes of play we plan to implement, the challenge is to break or “clear” blocks as rows of blocks are pushed into the play area. The blocks are pushed up from the bottom. It starts out (configurably) slow, and the rate that blocks are push gradually accelerates. In Endurance mode, the player must survive as long as possible. In versus mode, the players have to deal with the challenge of the accelerating blocks in addition to clearing garbage blocks dropped by the opponent.

The game play of Triforce looks rather simple to a player, but its inner workings are actually rather complex. The main game mechanics are combos and chains. The mechanics of combos are the easiest to explain. A combo is achieved by aligning at least three matching blocks together. However, you can combine sets of blocks by positioning things right before moving them. Below are some examples of possible combos.

*Figure 10: 3x Combo*

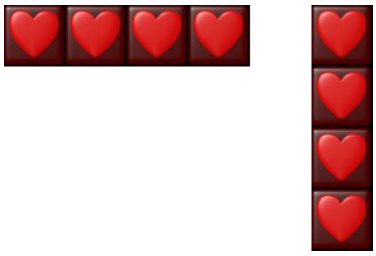


*Figure 11: 4x Combo*

*Figure 12: 6x Combo (2 sets of blocks)*

Chains are more difficult to pull off. Simple chains require a bit of planning, but it is only

possible to pull off large and complex chains with practice and skill. A chain is when at least one combo causes blocks to fall directly into another combo. The figures below feature some examples of chains in ascending difficulty.


*Figure 13: Basic Chain*

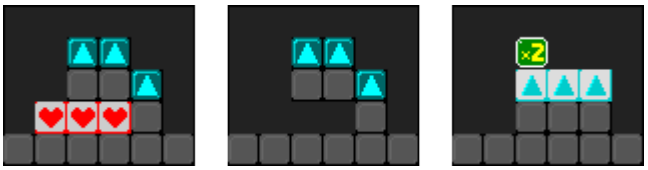



*Figure 14: Raised Skill Chain*

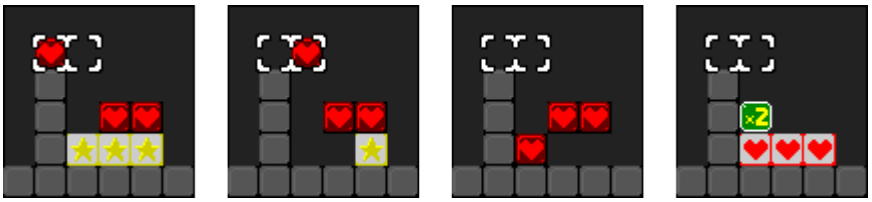



*Figure 15: Advanced Chain*

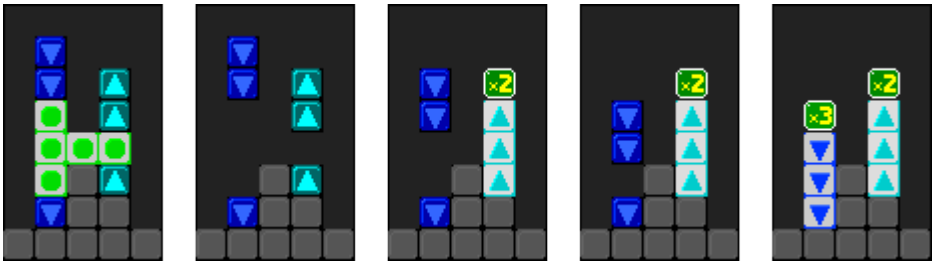



*Figure 16: Time-Lag Skill Chain*

Another important mechanic to cover is garbage blocks. Garbage blocks are formed in Versus mode when a player makes a chain. The amount of garbage blocks generated is directly related to the size of the chain. An example of garbage blocks can be seen in Figure 1. The garbage blocks are the set of blue blocks with an angry face in the middle. It was generated by the player on the left and dropped on the player on the right from above. When a combo is generated adjacent to a set of garbage blocks, all of the garbage blocks break and randomly generate new regular blocks.

Tools, Libraries, and Version Control:

We will be using Visual Studio 2010 as our development environment during the production Triforce. The game will require the GLUT library for graphics and sounds, and the 2DGraphics library. Due to some limitations of the 2DGraphics library, we will be adding several features to the header file. It will run in a Windows environment and compiled an x86 architecture. We are going to manage our source code and resources with Git and use Github to host our Git repository. We chose Git because both of us know the benefits of version control, and I have experience with Git. Github was chosen because of its many rich features for developers, such its bug/enhancement ticket system, commit tracking tools, wiki, and online code viewing.

## Glossary

Combo: A game mechanic where a player aligns at least three matching blocks

Chain: A game mechanic that requires a player to use smart timing and positining to cause combos to generate more combos from falling blocks.

Garbage Blocks: Blocks generated by a player in Versus mode by creating chains. Larger chains creater more garbage blocks.

OpenGL: An open-source, cross-platform API for creating 2D and 3D computer graphics.

GLUT (OpenGL Utility Toolkit): A closed-source procedural library written in C that contains utilities for OpenGL programs and input devices.

2DGraphics Library: A closed-source interface to GLUT created by Dr. Arif Wani that contains library functions useful for game development.

Version Control: Management of changes to documents, computer programs, or other collections of information.

Repository: A  data structure, usually stored on a server, that contains a set of files and directories, historical record of changes in the repository, set of commit objects, and a set of references to commit objects, called heads.

Git: A version control system that is fast, local, and distributed.

Github: A popular website for hosting Git repositories that is free to use.

Commit: A single revision in a revision control system.

Visual Studio 2010: An integrated development environment for Windows.

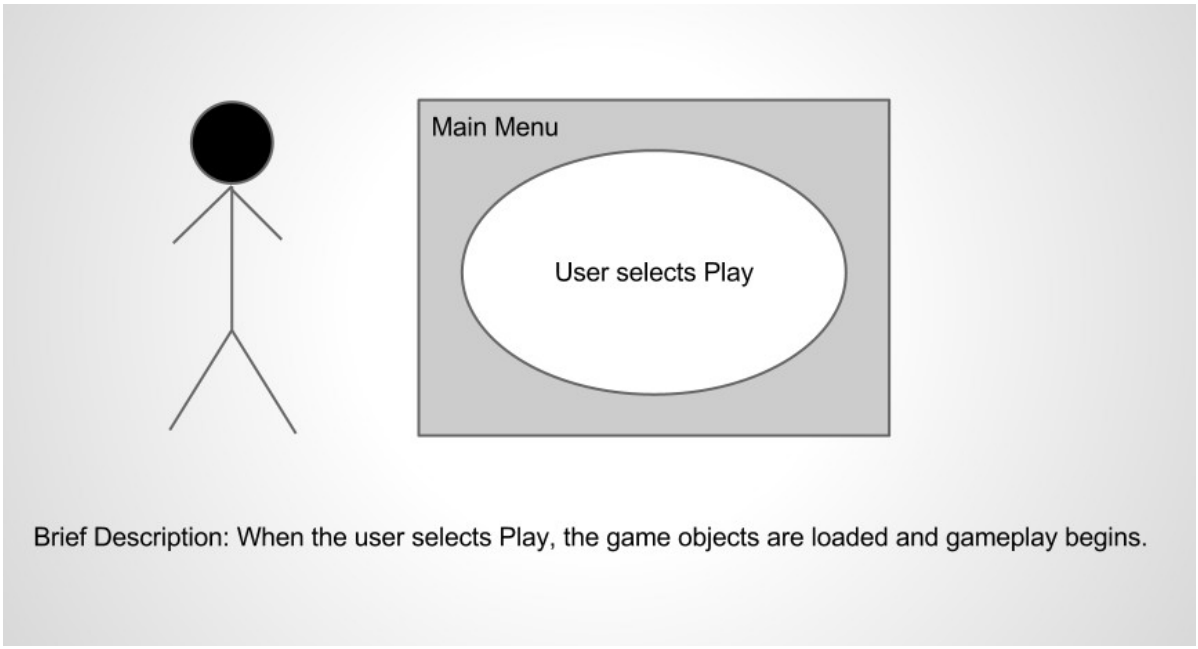
### 1.3.2 Initial Business Model

When a user runs the game, the first thing that happens is that the configuration file is loaded from disk. The configuration file contains key bindings for input, the preferred default theme, and theme directory location. After loading the configuration file, the initial menu is displayed to the user. The initial menu will display a background image and the buttons used to transition to other states of the game. The menu contains the following buttons: Play, Settings, Help, About, and Quit.

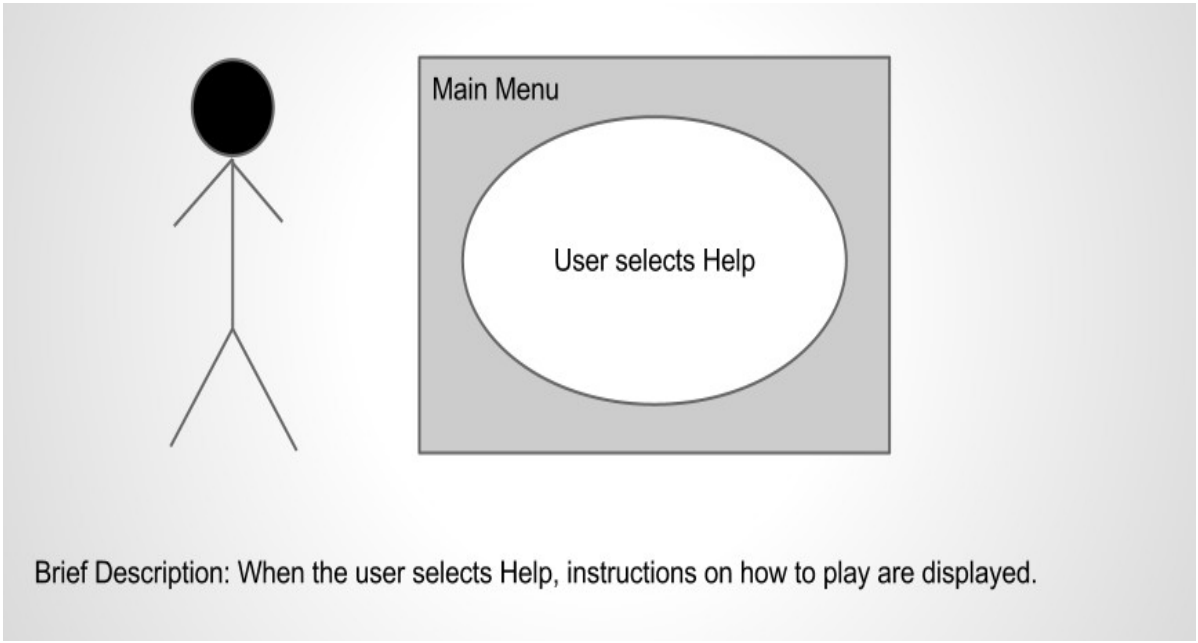
We plan to support multiple input devices. A single interface will control the mappings from input devices to a pseudo input device.

The actual gameplay will have several different states, including Play, Pause, Combo, and Countdown. Individual blocks will also need to have their own states, including Enabled, Disabled, Inactive, Combo, Falling.

## Use Cases

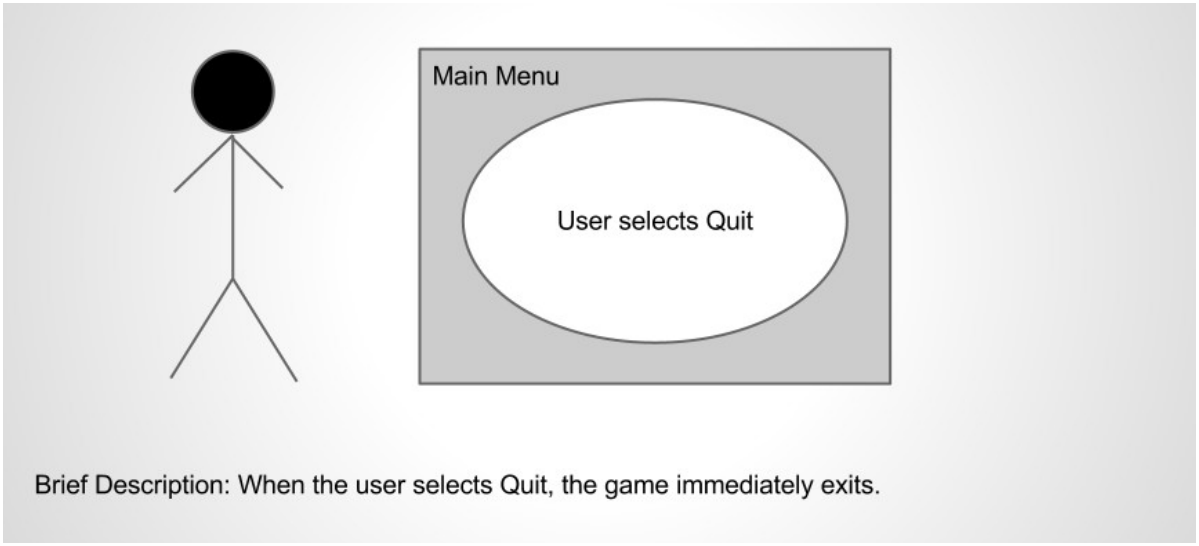



Brief Description: When the user selects Play, the game objects are loaded and gameplay begins.

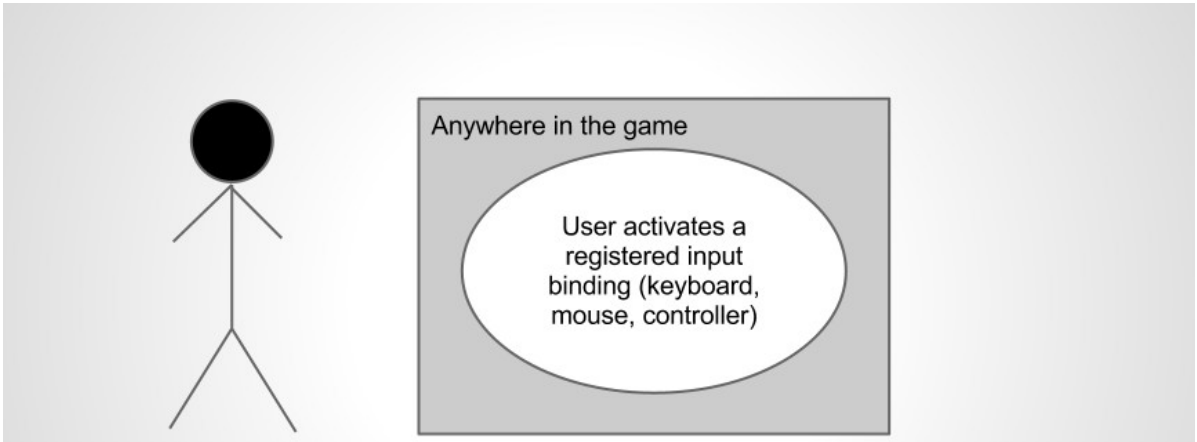



Brief Description: When the user selects Help, instructions on how to play are displayed.

## Use Cases

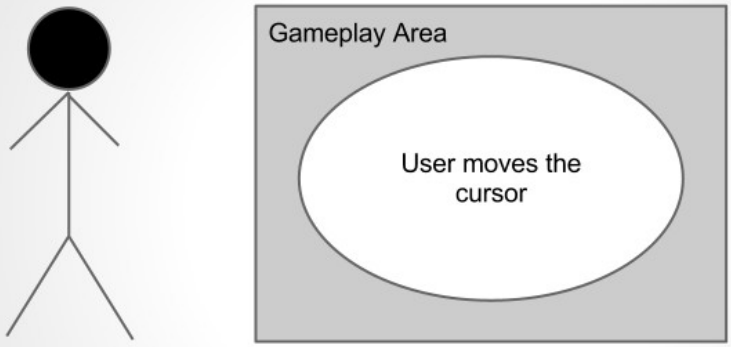



Brief Description: When the user selects Quit, the game immediately exits.

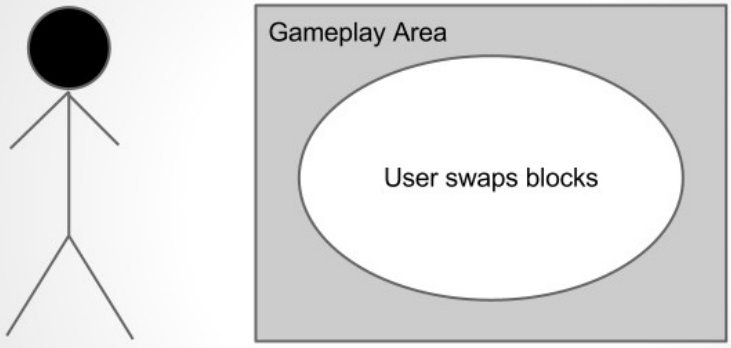



Brief Description: When the user activates a registered input binding, such as pressing/holding a key, clicking the mouse, or pressing a button on an xbox controller, its registered callback function is called.

## Use Cases

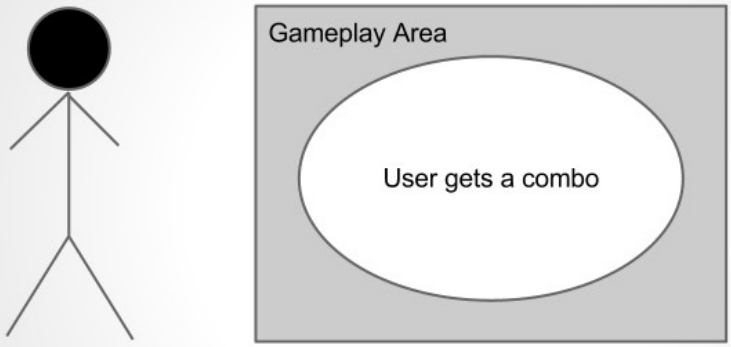



Brief Description: When the user moves the cursor, the cursor's position is updated one block in the direction the user requests, and the graphic is redrawn.

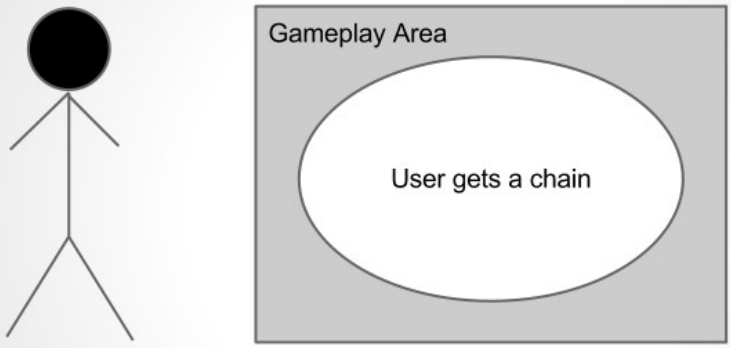



Brief Description: When the user swaps the two blocks, the graphics and states of the selected blocks are swapped with each other.

## Use Cases

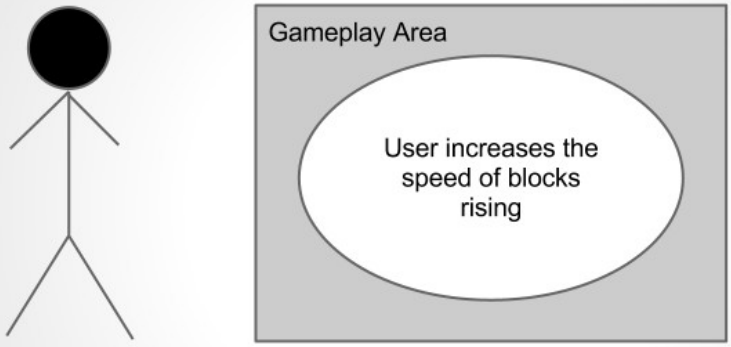



Brief Description: When the user gets a combo, blocks will temporarily stop rising. An animation is played, and then the blocks break. The blocks above the cleared combo will fall down.

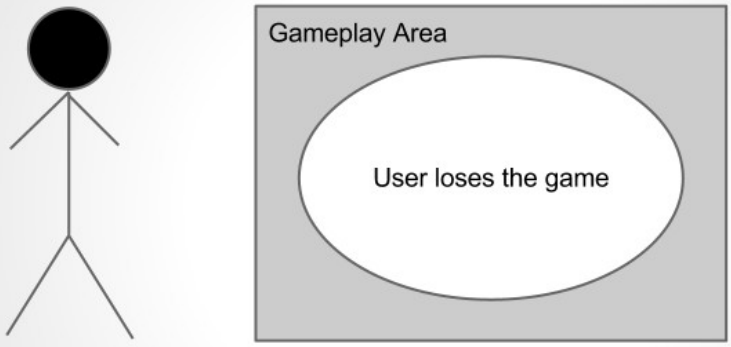



Brief Description: When the user gets a chain, it will act just like a regular combo. However, a bonus will be displayed to indicate that a chain occurred.

## Use Cases




Brief Description: The user can activate a key binding that forces blocks to rise at a faster rate. This rate will be constant, and the current speed will be temporarily replaced with this constant while the binding is activated.




Brief Description: The user loses the game when a block hits the top of the grid.

### 1.3.3 Obtain Initial Requirements

The user begins the game and is presented with the main menu. The options are Play, Help, and Quit. The user begins the game by selecting Play. A tutorial is displayed if he selects Help. The game exits if he selects Quit. The game must support multiple input devices, so multiple bindings must map to a single action.

The user has a limited amount of actions. While in game, the user can:

- move in the cursor up, down, left, and right
- temporarily increase the speed that blocks rise into the play area
- swap blocks

When the user swaps blocks, it has a possibility of causing a combo, thus changing the state of the game.

### 1.3.4 Obtain Detailed Requirements

Input:

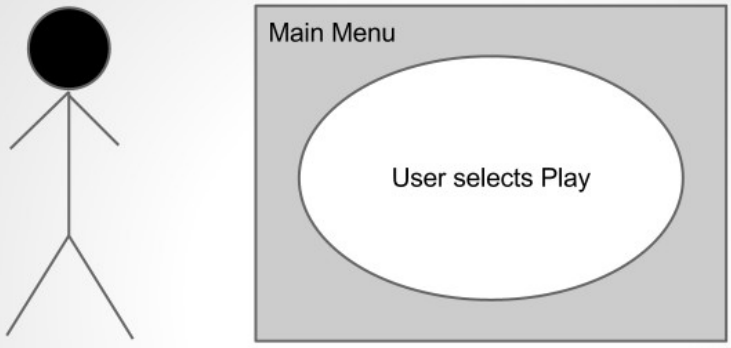
- Since multiple input devices must be supported, a generic interface must be able to handle all of them.
- Different states must be recognized, such as when a binding is pressed, held down, and released.
- More details in Brandon's report

Game Mechanics:

- The cursor moves one space at a time.
- The cursor cannot move outside of the grid.
- When a block comes into contact with the top of the grid, the player loses the game.
- A timer will keep track of the time played and be displayed on the screen.
- The score will be displayed on the screen.
- Chains must be kept track of between combos and fall events.
- At least 3 blocks form a combo, but it is possible to have up to 14 in a single combo.
- After a combo clears, an algorithm must detect which blocks need to fall down.
- When a falling block lands, an algorithm must detect if a combo occurs.
- When a combo >=4 or a chain occurs, it shall display an animated bonus where the event happened.
- Combos will multiply the number of blocks in the combo by a constant and add it to the score.
- Chains will act as multipliers to the combo that occurs and drastically increase the score.

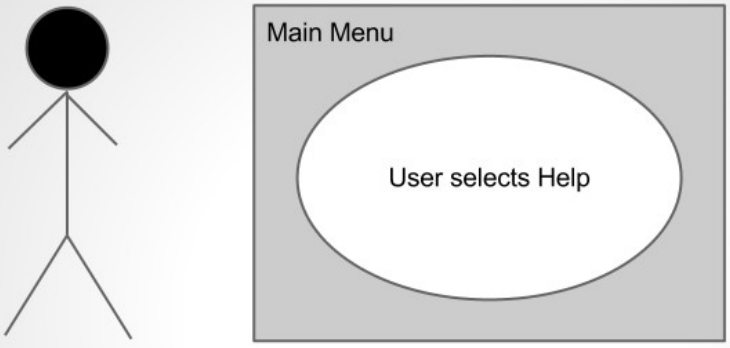
## 1.3.5 Obtain Detailed Requirements

### Use Cases




Updated Description: When the user selects Play, the game state is changed. The grid, cursor, blocks, timer, and score are initialized.

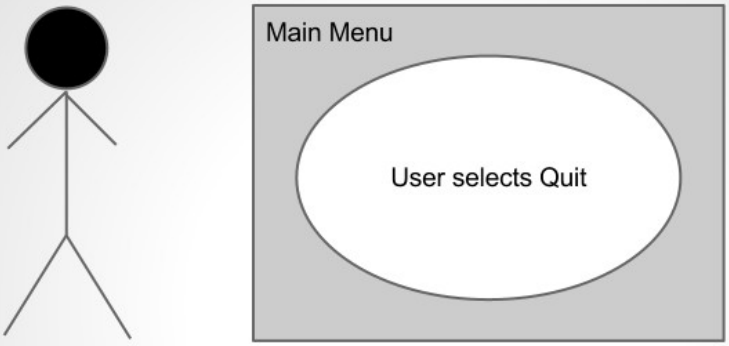
Step-by-step:
1. User selects Play
2. Change game state to PLAY
3. Initialize grid, cursor, blocks, timer, and score




Updated Description: When the user selects Help, instructions on how to play are displayed.

Step-by-step:
1. User selects Help
2. Display text/graphics at the correct position that explain how to play.

## Use Cases



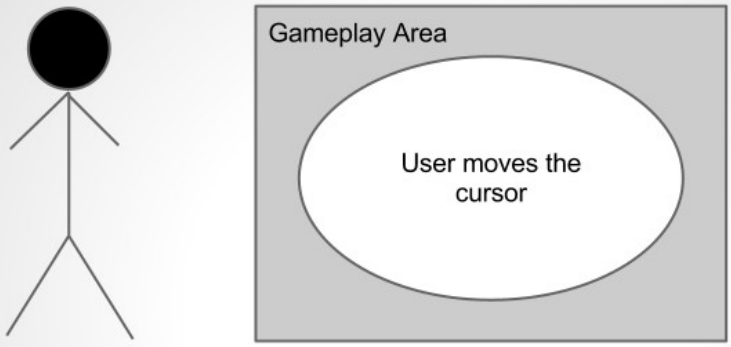

Updated Description: When the user selects Quit, the game immediately exits.

Step-by-step:
1. User selects Quit
2. Game exits


Updated Description: When the user activates a registered input binding, and the binding is in a registered state (e.g. on_press, is_down, _on_release), its registered callback method is invoked.

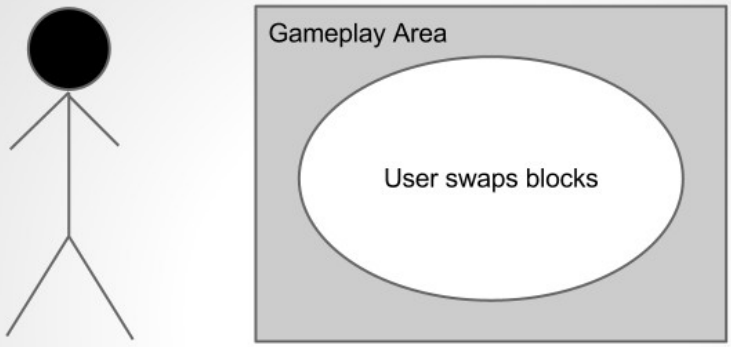
Step-by-step:
1. Map a keybinding to an action and an input state
2. User activates keybinding
3. Callback method is invoked

## Use Cases




Updated Description: When the user moves the cursor, the cursor's position is updated one block in the direction the user requests, and the graphic is redrawn.

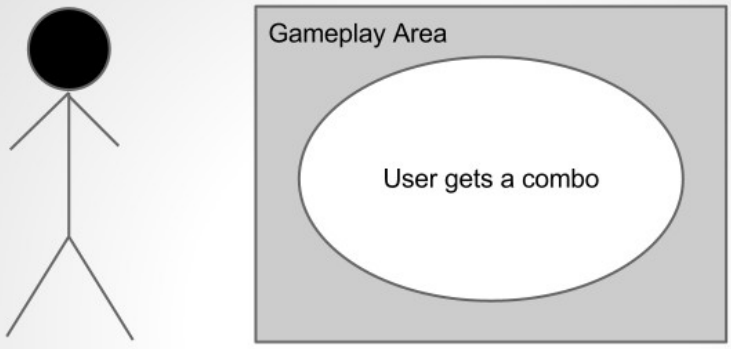
Step-by-step:
1. User input calls cursor move method.
2. Calculate cursor's new coordinates based on the direction it is moved.
3. Do not let cursor escape bounds of grid.
4. Redraw the cursor.




Updated Description: When the user swaps the two blocks, the graphics and states of the selected blocks are swapped with each other. Combo detection and fall detection algorithms are executed.

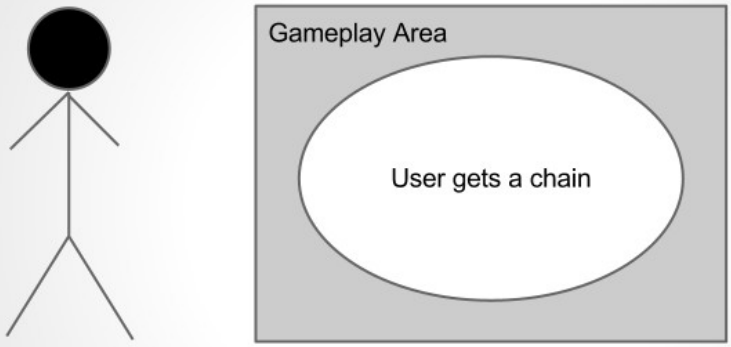
1. User input calls swap block method.
2. The graphics and states of the selected blocks are swapped with each other.
3. Detect combos for both blocks.
4. Detect falls for both blocks

## Use Cases




Updated Description: When the user gets a combo, the blocks in the combo go into a combo state. All blocks will temporarily stop rising. A combo with 4 or more blocks displays a bonus animation. The blocks in the combo display a special effect. Detect if any blocks need to fall.

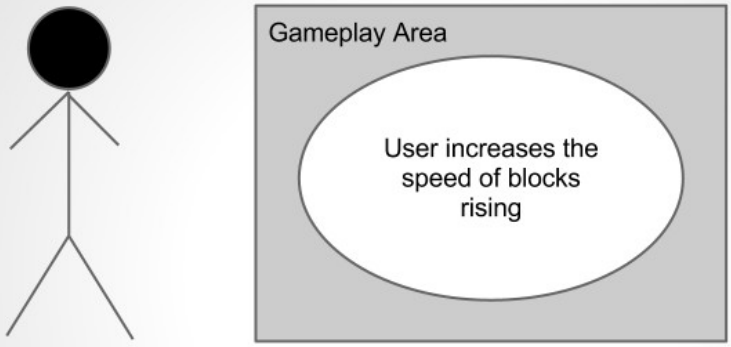
1. Combo is detected
2. Set state of Blocks in the combo to COMBO and state of Grid to COMBO
3. Display animated bonus if combo has 4 or more blocks in it.
4. Display different graphic while blocks are in COMBO state and pause the rising of new blocks while grid is in COMBO state
5. After time for combo has elapsed, set state of blocks in combo to DISABLED, and Grid state to PLAY
6. Detect blocks that need to fall




Updated Description: When the user gets a chain, it will act just like a regular combo. However, a bonus will be displayed to indicate that a chain occurred.

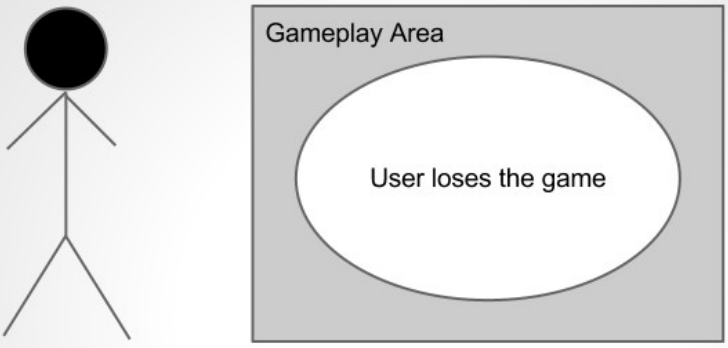
Step-by-step:
Same as combo, except for step 3.
3. Display animated bonus if combo has 4 or more blocks in it, and display an animated bonus for the chain.

## Use Cases




Updated Description: The user can activate a key binding that forces blocks to rise at a faster rate. This rate will be constant, and the current speed will be temporarily replaced with this constant while the binding is activated.

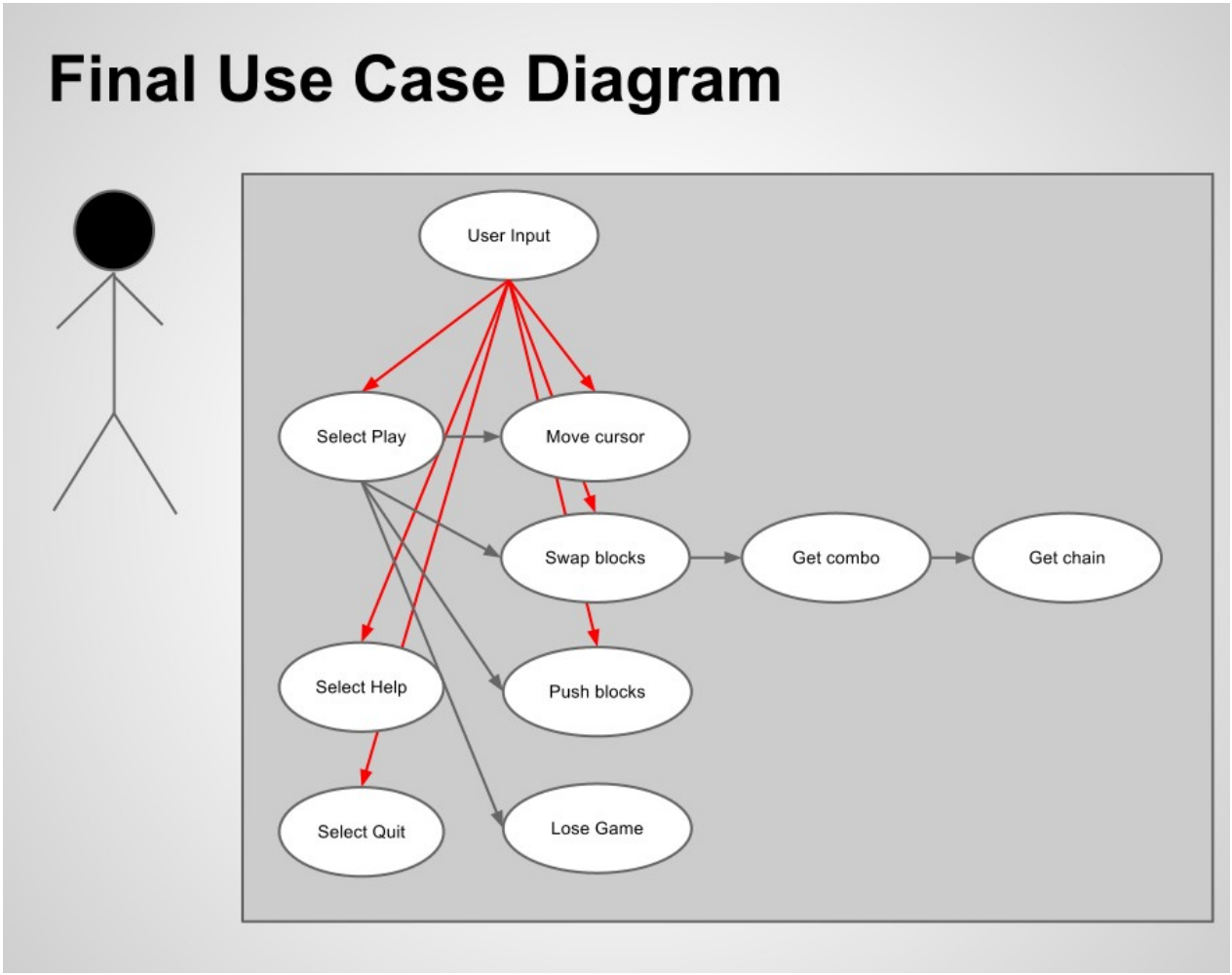
1. User input invokes method to make blocks rise faster.
2. Grid's state changed to PUSH
    - while in PUSH state, automatic rising of blocks is disabled.
3. Blocks are pushed up by 1 pixel on a short, constant interval while the input binding is active.
4. When user releases the button, the Grid's state is changed to PLAY.




Brief Description: The user loses the game when a block hits the top of the grid.

1. Block hitting the top of the Grid is detected.
2. Grid state changes to GAMEOVER.
    - While in GAMEOVER state, blocks stop pushing up.
3. Initialize a countdown timer.
4. Display graphics for game over screen.
5. Go back to main menu.

**Use Cases**


Final Use Case Diagram
User Input
Select Play
Move cursor
Swap blocks
Get combo
Get chain
Select Help
Push blocks
Select Quit
Lose Game

# 2.1 Analysis Workflow

### 2.1.1 Functional Modeling

**Normal scenarios**

1. Game starts and shows the main menu
2. Player selects the Play button
    - 2.1. Game is loaded and customized based on selected settings
    - 2.2. Timer counts down from 3 then starts the game
    - 2.3. Player moves with cursor
    - 2.4. Player swaps blocks
    - 2.5. Execute the combo detection subroutine
    - 2.6. If a combo is detected:
        - a. Grid changes state to COMBO
        - b. Blocks in the combo are changed to state COMBO
        - c. After S seconds the blocks break
        - d. Execute the fall detection subroutine (goto 2.7a)
    - 2.7. If no combo is detected:
        - a. Execute the fall detection subroutine
        - b. If a fall is detected:
            1. Set the blocks that need to fall to the FALL state
            2. Make the blocks fall by Y pixels every S milliseconds until they land on a non-disabled block.
            3. Execute the combo detection subroutine on each FALL block
            4. Increment the chain counter for each combo detected and goto 2.6
            5. Set the chain counter to 0 if no combo detected
            6. Set Grid's state to PLAY
    - 2.8. Player selects the Pause button
        - a. The game's state changes to PAUSE
        - b. The blocks are not displayed to prevent cheating
        - c. The player selects the Pause button again
        - d. The game's state changes to PLAY
        - e. Normal gameplay resumes

2.9. Player selects the Menu button

a. Game objects are deleted

b. The player is returned to the main menu

2.10. Player selects the Quit button

a. The game immediately exits

3. Player selects the Settings button

3.1. The settings menu is loaded

3.2. Player toggles sound on/off

3.3. Player toggles mouse on/off

3.4. Player changes key bindings

3.5. Player clicks the Save button

a. Settings are saved

b. Main menu is loaded

4. Player selects the Help button

4.1. Some text and graphics are displayed that explain how to play the game

4.2. Player selects the Back button

4.3. Main menu is loaded

5. Player selects the About button

5.1. A short description of the game, credits, and licensing information is displayed on the screen.

5.2. Player selects the Back button

5.3. Main menu is loaded

6. Player selects the Exit button

6.1. Game immediately exits.

**Exception scenarios**

Early Quit:

1. Player starts playing the game
2. Player clicks Quit
3. Game immediately exits

Player intentionally loses

1. Player holds the push-row button
2. The blocks hit the top very quickly
3. Player loses
4. Game over screen is displayed
5. Player returns to main menu

Player gets very large chains

1. The player is skilled enough to chain many combos together
2. The game is able to handle it because combos are detected after every combo occurs.

## 2.1.2 Entity Class Modeling

**Noun extraction**

When the **game** is loaded, the **configuration file** is read and saved into **settings**. Then the **player** is presented with the **main menu**. The **player** can click **buttons** to take them to other **menus** or to play the **game**. The **settings menu** lets the user configure **options**. **Options** are saved to a **configuration file**. The **help menu** displays explains how to play the **game**. The **About menu** displays gives some background information about the **game**. The **player** clicks the **play button** and to start the **game**.

The **game play area** consists of the of a **grid** of **blocks**. The **player** controls a **cursor** and can use it to swap **blocks**. If at least 3 **blocks** are lined up, a **combo event** occurs. If there are disabled **blocks** under an enabled **block**, then a **fall event** occurs. A **fall event** can only occur after the player swaps or a **combo event.** A **combo event** can only occur after the **player** swaps or a **fall event**. The **grid controller** is responsible for detecting **combo events** and **fall events**. A **heads-up display** (**HUD**) keeps track of the **time** and displays a **timer** and the **score**.

**Input devices** all go through a single **input controller** that controls the flow of execution based on the current **state** of the **game** and the **action** bound to the **input event**.

Identification of nouns: game, player, main menu, settings menu, help menu, about menu, button, settings, options, game play area, grid, blocks, combo event, fall event, grid controller, HUD, timer, score, input device, input controller, state, action, input event

Entity classes:

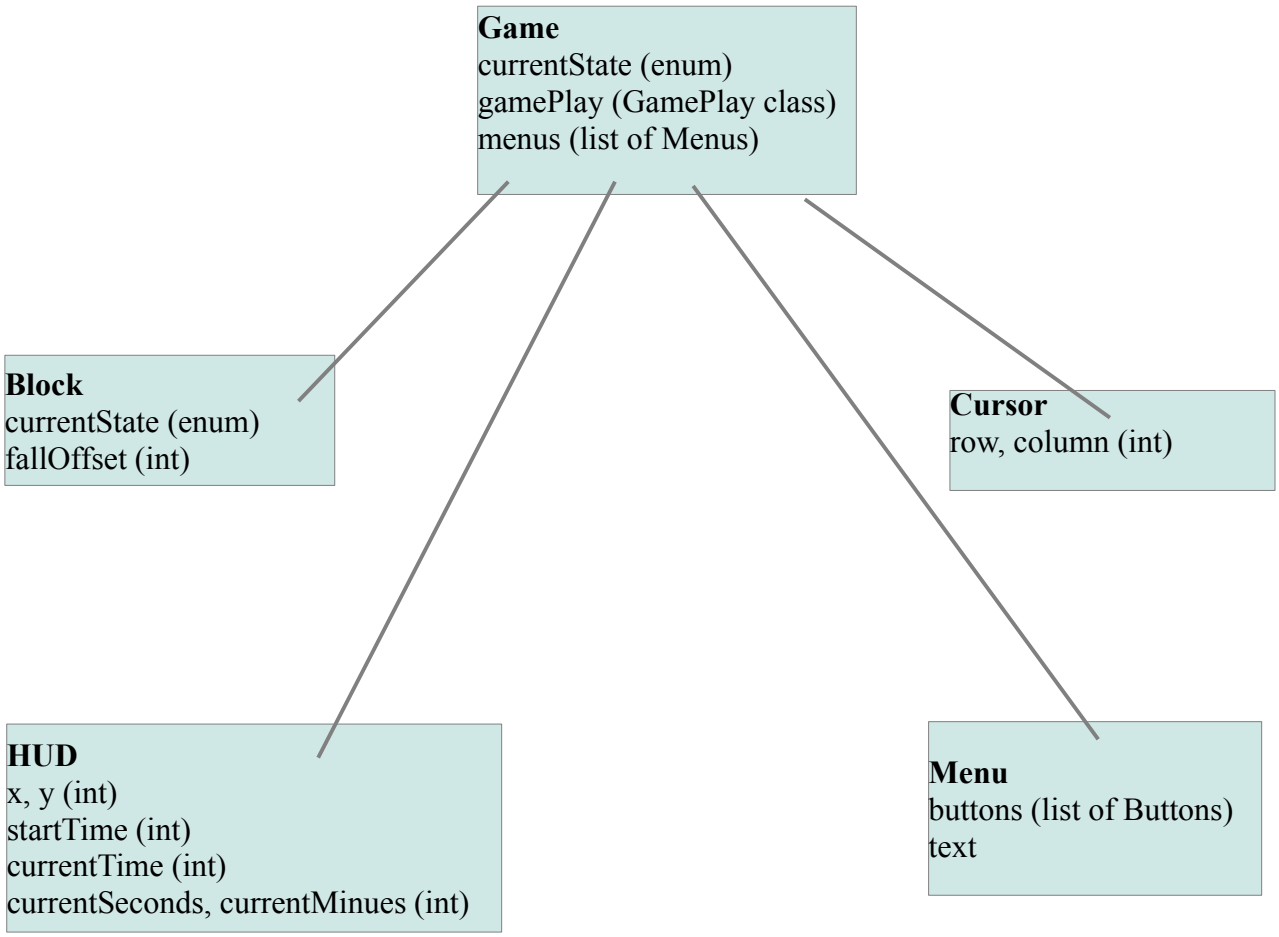
Game

HUD

Block

Cursor

Menu



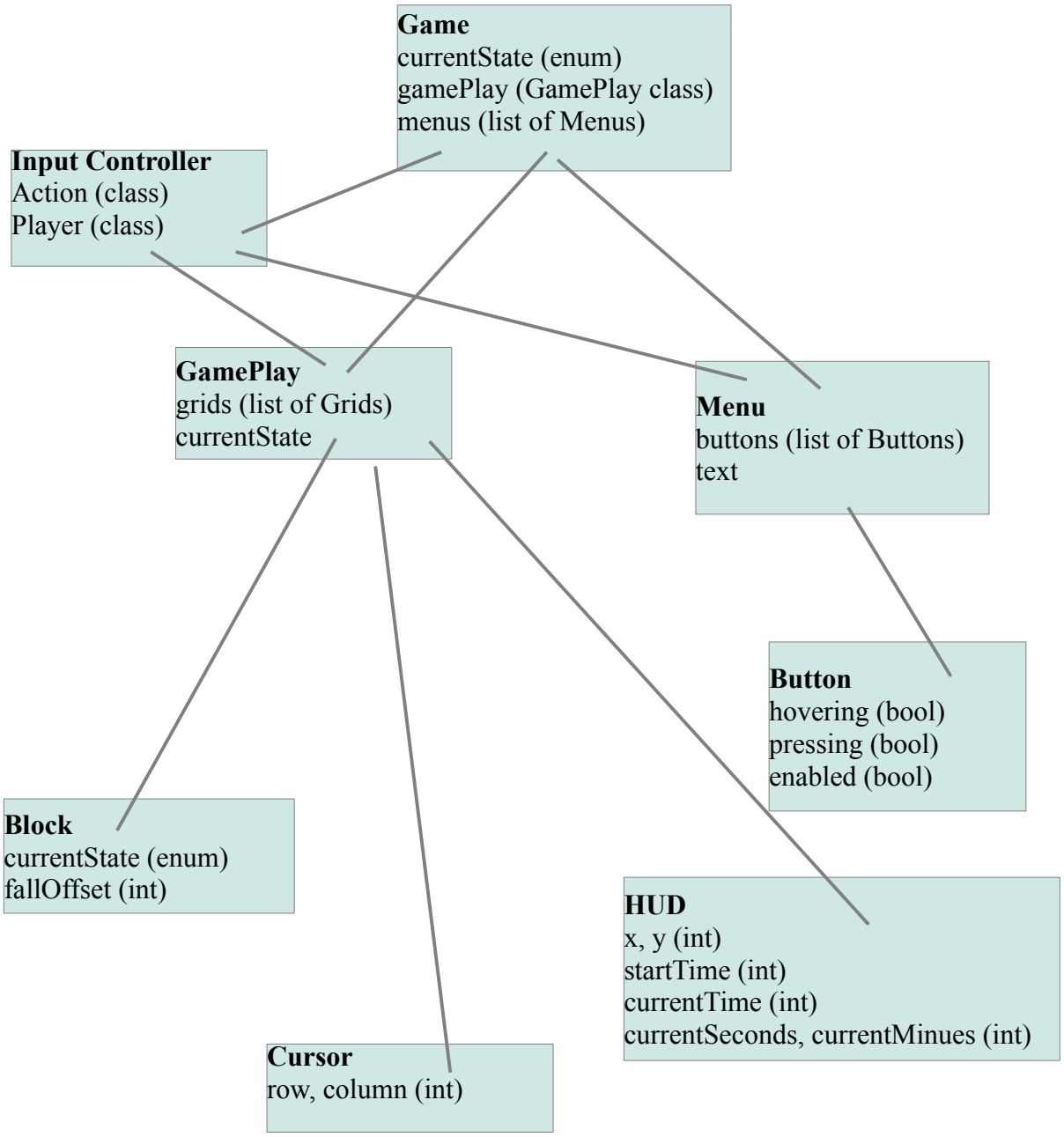
### 2.1.3 Boundary Class Exaction

Boundary classes:

Input

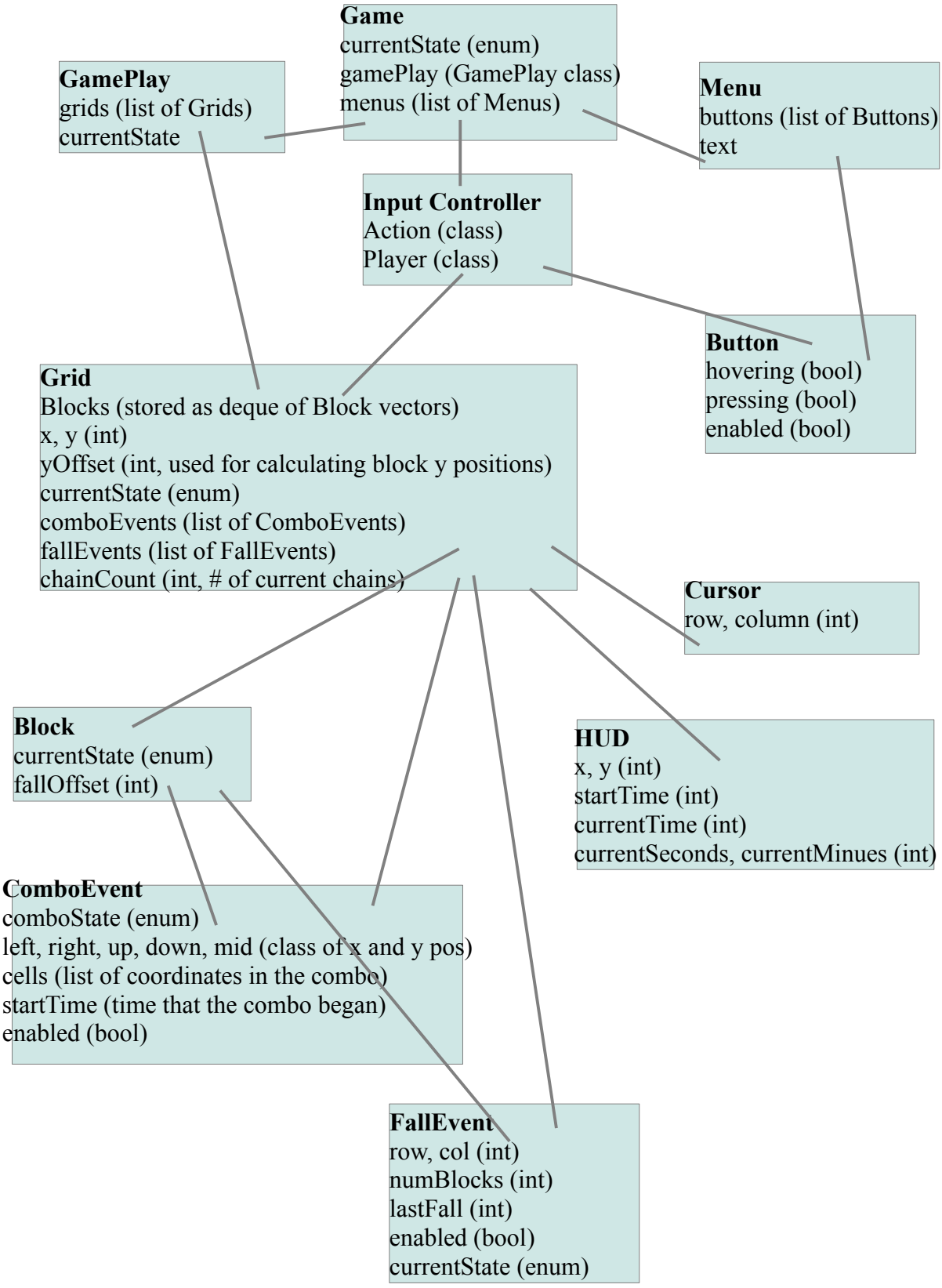
Button

GamePlay

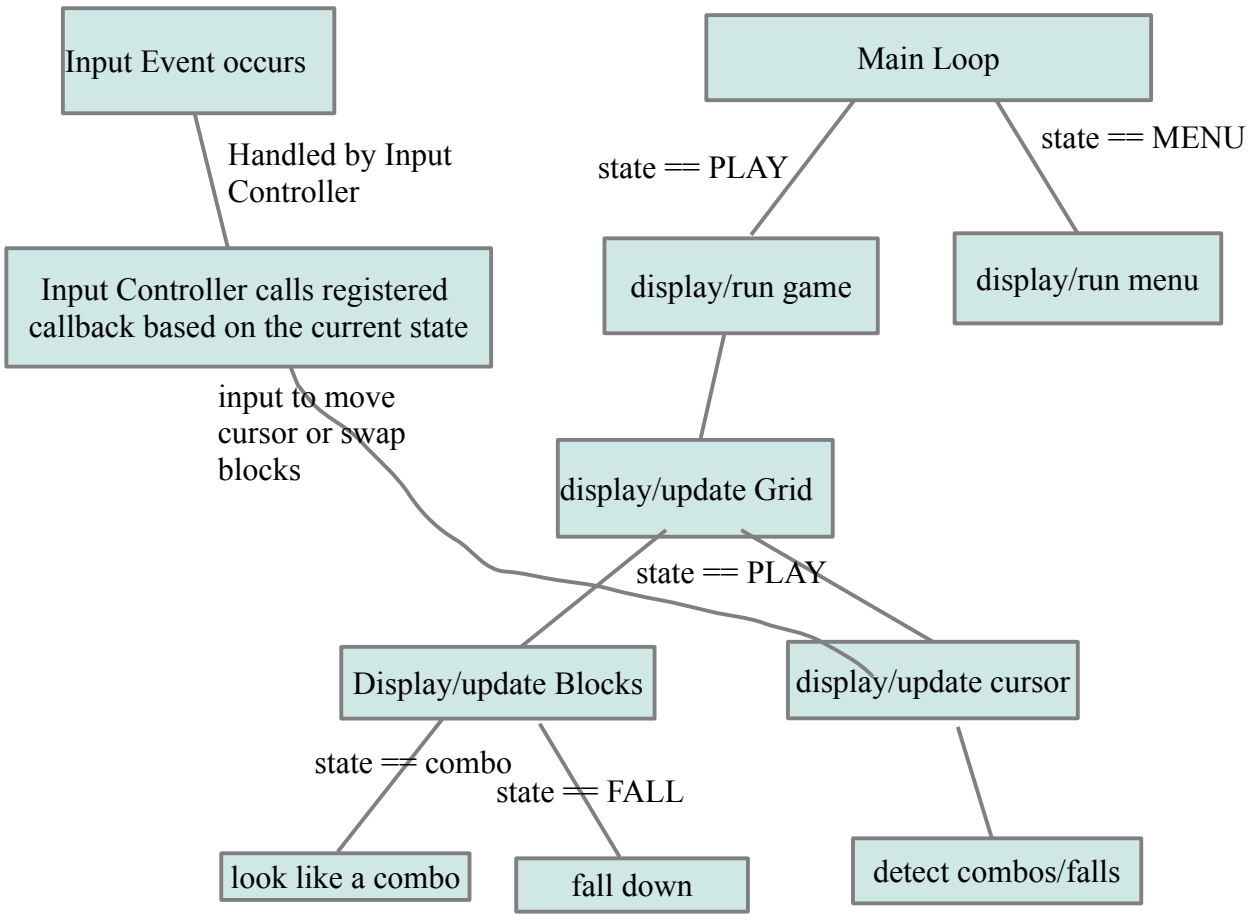


### 2.1.4 Control Class Modeling

Control classes: Grid, ComboEvent, FallEvent



### 2.1.5 Dynamic Modeling



### 2.1.6 Communication Modeling

## 2.2 Walkthrough

It seems like some attributes are missing. Re-evaluate your classes and consider what else should be added.

Make sure to include all the states in your dynamic model.

Break functional model up into different scenarios

Use UML conventions

- Walkthrough by Brandon Hinesley

# 2.3 Revised Analysis after Walkthrough

## 2.3.1 Functional Modeling

### Normal Scenarios

Scenario 1:

1. Player selects Play
2. Change game's stay to PLAY
3. Initialize the Grid with randomized blocks, and initialize the timer and score.

Scenario 2:

1. Player selects Help
2. Display a helpful guide for how to play
3. Display a button to go back

Scenario 3:

1. Player selects Quit
2. Immediately exit the game

Scenario 4:

1. Player moves cursor
2. Cursor's position is recalculated based on the direction it is moved
3. Cursor is redrawn

Scenario 5:

1. Player swap blocks
2. The selected blocks are swapped with each other
3. Detect combos for each block
4. If combo is detected, goto combo detected scenario
5. Detect fall for each block
6. If fall is detected, goto fall detected scenario

Scenario 6:

1. A combo is detected
2. Increment the chain count for the combo
3. If combo has 4 or more blocks, display an animated bonus with the number of blocks in it
4. Set state of Grid and Blocks in the combo to COMBO
5. Remain in the COMBO state for (N number of blocks * T milliseconds)
6. Set the state of the COMBO blocks to DISABLED when the timer elapses, and set Grid's state to PLAY
7. Detect falling blocks
8. Group all falling blocks in the same fall event, and goto scenario for detecting a fall event

Scenario 7:

1. A fall event is detected
2. Set all the blocks in the fall event to the FALL state
3. For each block, fall Y pixels every T milliseconds while in the FALL state
4. Set block's state to ENABLED when it lands in a cell above another enabled block.
5. Detect combo for each falling block
6. If combo is detected, goto combo detected scenario

Scenario 8:

1. Player input requests to push blocks up faster
2. Grid set to PUSH state
3. Automatic block rising is temporarily disabled
4. Blocks are pushed up every T milliseconds, an interval less than the automatic interval
5. Grid restored to PLAY state when input binding is no longer active

Scenario 9:

1. A block comes into contact with the top of the Grid
2. Game set to GAMEOVER state for T milliseconds
3. Game over graphic displayed on screen while in GAMEOVER state, and player controls are disabled
4. Go back to main menu when timer elapses

**Exception Scenarios**

Scenario 1:

1. Player sends input that is not a registered binding
2. Nothing happens

## Identification of Nouns

When the **game** loads, the **player** is presented with the **main menu**. The **player** can select **buttons** that take **actions** when selected. The **buttons** on the **main menu** are **Play**, **Help**, and **Quit**. The **Help menu** displays explains how to play the **game**. The **Quit** button closes the game. The **player** clicks the **Play button** and to start the **game**.

The **game play** consists of a **grid** for each player. A **grid** contains **blocks** and a **cursor**. The **player** controls the **cursor** and can use it to swap **blocks**. If at least 3 **blocks** are lined up, a **combo event** occurs. If there are disabled **blocks** under an enabled **block**, then a **fall event** occurs. A **fall event** can only occur after the player swaps **blocks** or after a **combo event.** A **combo event** can only occur after the **player** swaps or after a **fall event**. The **grid** is responsible for detecting **combo events** and **fall events**. An animated **bonus** is displayed when the player gets a big **combo** or a **chain**. A **heads-up display** (**HUD**) displays a **timer** and the **score**.

**Input devices** all go through a single **input controller** that controls the flow of execution based on the current **state** of the **game** and the **action** bound to the **input event**.

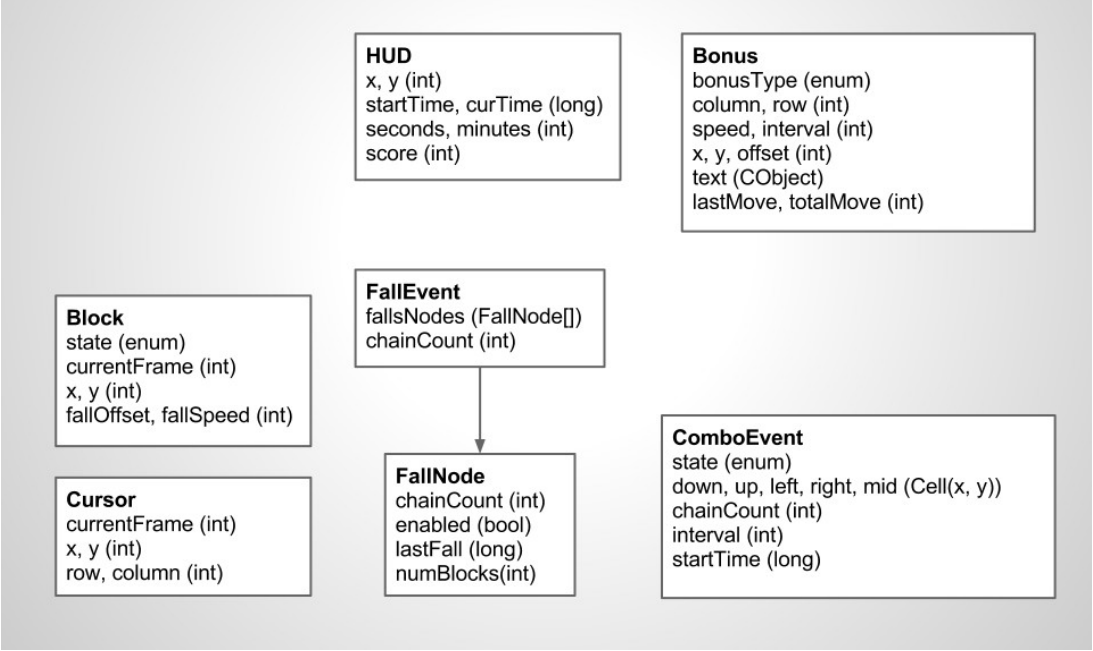
## Noun Extraction

player, menu, button, actions, grid, block, cursor, combo event, fall event, bonus, HUD, timer, score, input device, input controller, state

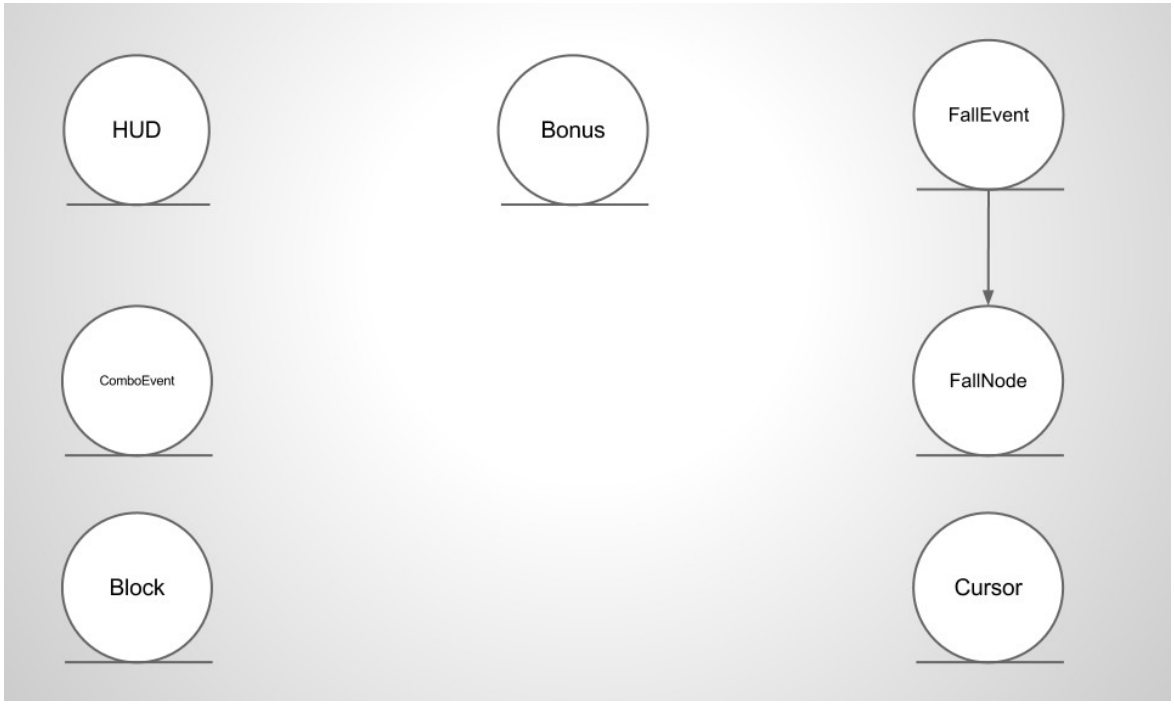
## 2.3.2 Entity Class Modeling

### Noun Extraction

button, block, cursor, combo event, fall event, bonus, HUD, timer, score




*Attribute Box Diagram*

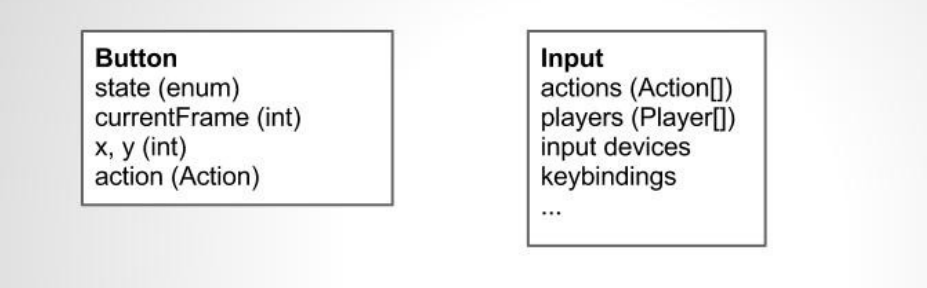



*UML Class Diagram*

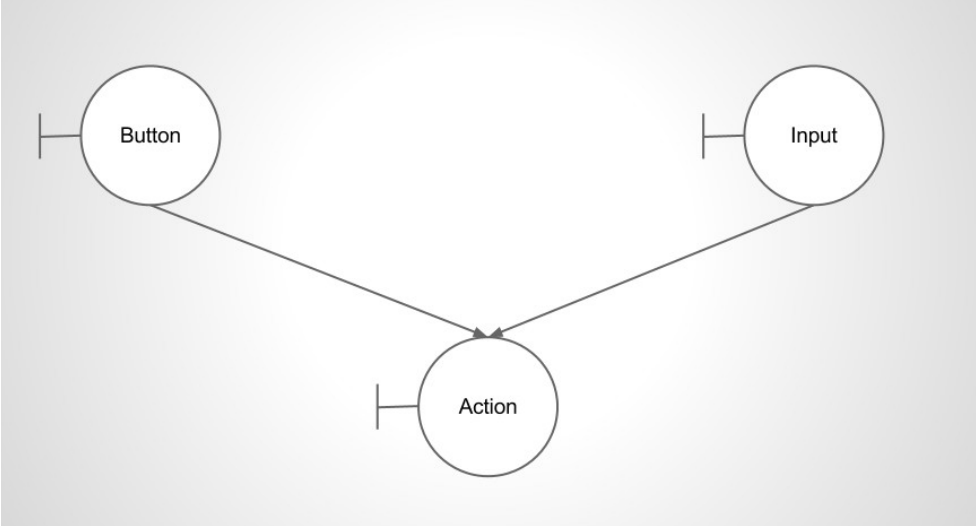
## 2.3.3 Boundary Class Modeling

**Noun Extraction**:

button, player, input, input device, action




*Attribute Box Diagram*

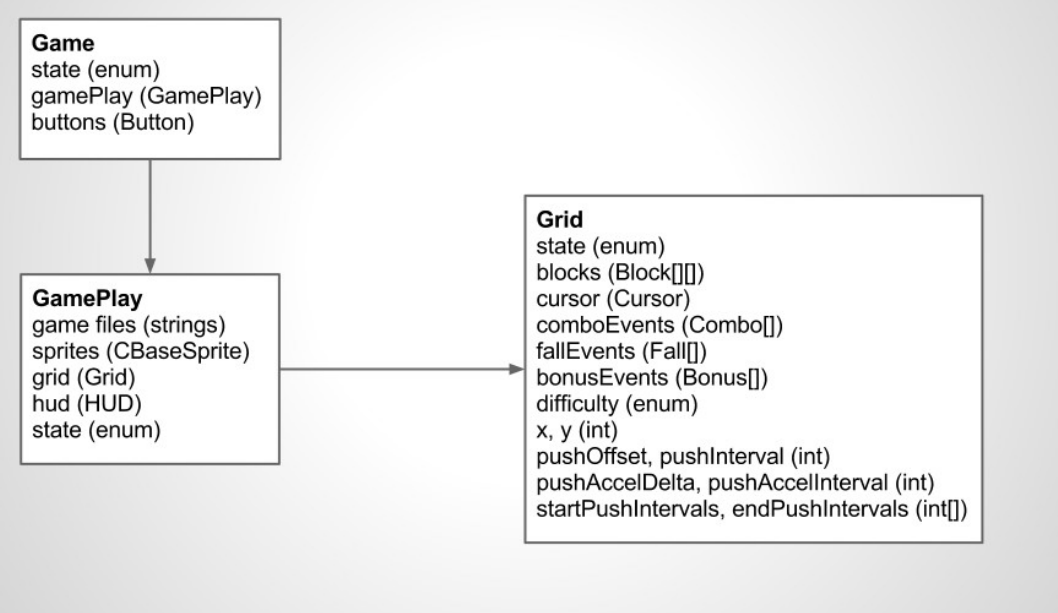



*UML Class Diagram*

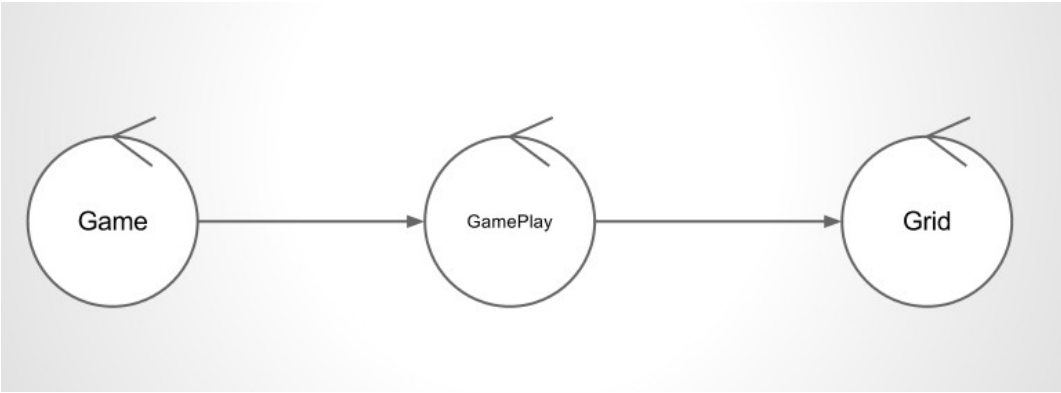
## 2.3.4 Control Class Modeling

**Noun Extraction:**

game, game play, grid

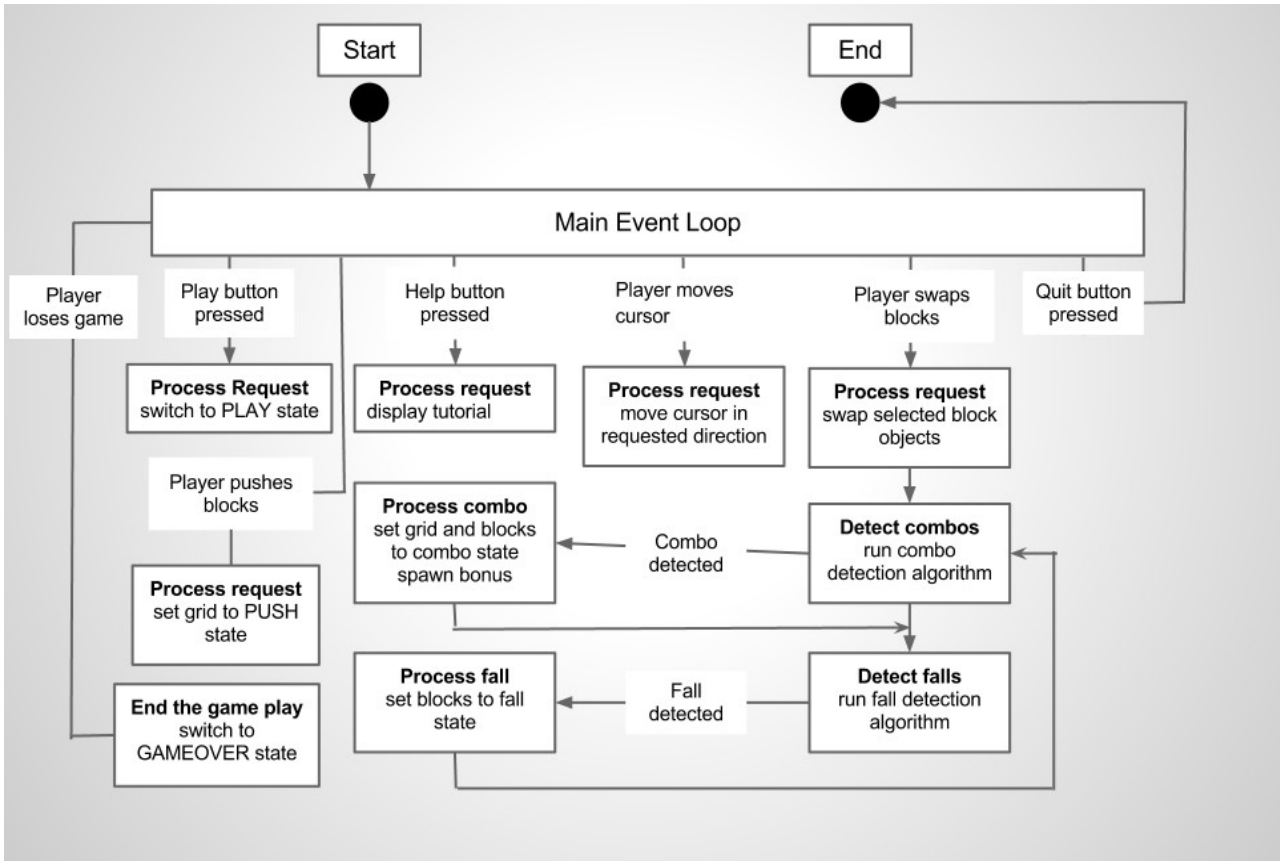



*Attribute Box Diagram*



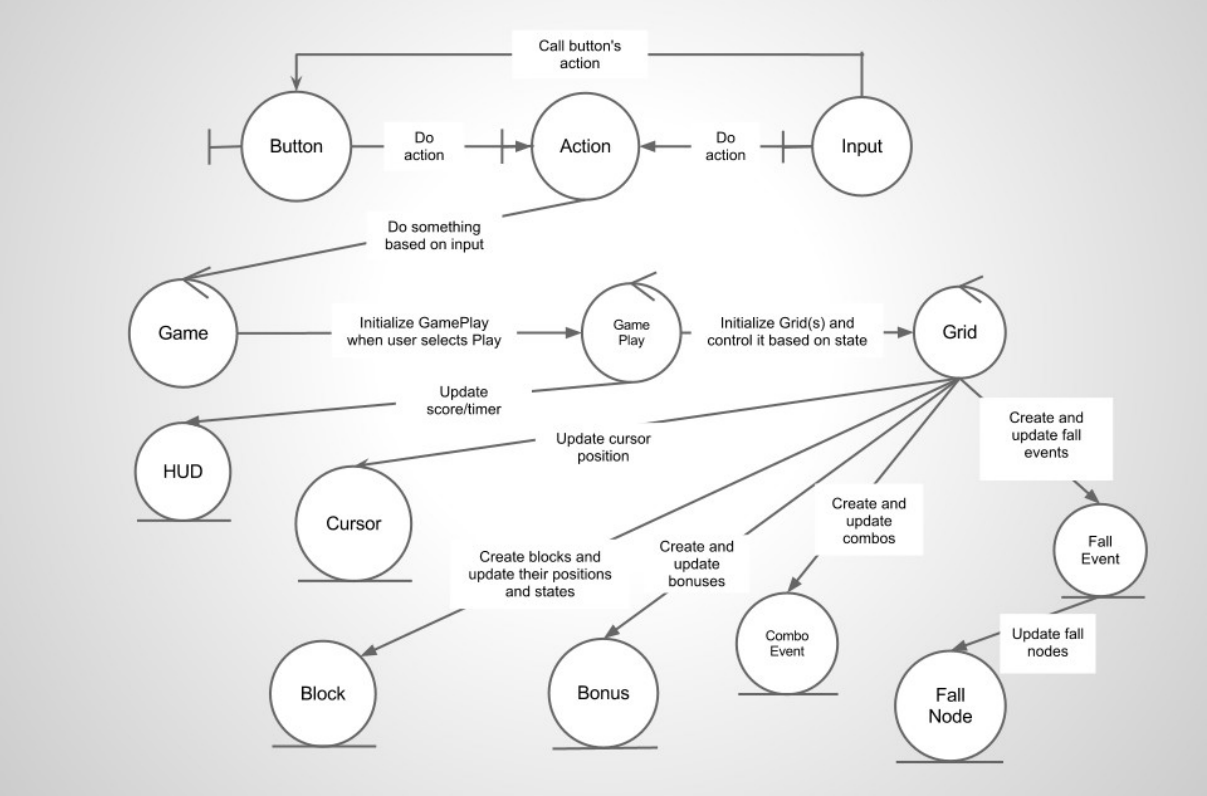

*UML Class Diagram*

## 2.3.5 Dynamic Modeling




*Dynamic Model Diagram*

## 2.3.6 Communication Modeling




*Communication Model Diagram*

# 3.1 Design Workflow

## 3.1.1 Component Design

CRC Cards:

**Class:** Triforce

**Responsibility:**

Load the main menu
Load the configuration file
Pass control of the program based on the current state
Set up input handlers

**Collaboration:**

Menu
GamePlay
Input

**Class:** Input

**Responsibility:**

Be the middle-man between supported input devices and the class that handles actions
Provide an interface for all classes to register actions based on input received and current state

**Collaboration:**

Triforce
GamePlay
Menu
Grid
Cursor

**Class:** Menu

**Responsibility:**

Display menu buttons
Handle input
Set settings

**Collaboration:**

Button
Input
Triforce

**Class:** GamePlay

**Responsibility:**

Initialize the game
Handle input that affects GamePlay (such as pausing)
Load objects that will be displayed during the game

**Collaboration:**

Cursor
Block
Combo
Fall
Bonus
Input

**Class:** Grid

**Responsibility:**

Provide an interface to GamePlay for control the cursor
Update the Cursor's coordinates
Swap Blocks
Detect Combos
Detect Falling Events
Display Bonuses
Keep track of all events and update them for each iteration of the main loop
Call display methods for all objects that must be displayed
Calculate player's score

**Collaboration:**

Cursor
Block
Combo
Fall
Bonus
Input

**Class:** Cursor

**Responsibility:**

Display itself
Provide methods to move the cursor
Keep track of position

**Collaboration:**

Input
Grid

**Class:** Block

**Responsibility:**

Display itself
Provide methods for swapping Blocks
Provide methods for falling down

**Collaboration:**

Grid
Combo
Fall

**Class:** Combo

**Responsibility:**

Provide an interface to Grid for managing combo events
Update the state of the Combo
Update the Grid's state based on the state of the Combo
Keep track of chains
Detect fall events when the combo has elapsed

**Collaboration:**

Grid
Block
Fall

**Class:** Fall

**Responsibility:**

Provide an interface to Grid for managing fall events
Update the state of the fall
Manage individual FallNodes, calling their update functions for every iteration of a Fall
Keep track of chains
Detect combo events when the fall has elapsed

**Collaboration:**

Grid
Block
Combo

**Class:** FallNode

**Responsibility:**

Provide an interface to Fall for managing a single list column of falling blocks
Update state of FallNode
Keep track of position and number of Blocks in the fall
Cause Blocks to fall for each iteration

**Collaboration:**

Fall
Grid
Block

**Class:** HUD

**Responsibility:**

Display the score
Keep track of and display the time

**Collaboration:**

GamePlay
Grid

**Class:** Button

**Responsibility:**

Display itself based on current state
Register and call callback functions when pressed

**Collaboration:**

GamePlay
Grid

### 3.1.2 Interface Design

The currently displayed objects are controlled by the state of the game. States are defined as enums that contain all possible states. Different functions get called based on the state of the game. For example, when the game is in the Play state, the GamePlay's display() and composeFrame() are called, which call Grid's respective functions. Grid calls Block, Cursor, and Bonus display() and composeFrame() functions.

| Interface Actions | Interface Objects | User Actions (events) |
|---|---|---|
| display() | Calls display() methods for each class, starting at the base and branching out in a tree fashion. Every object that is actually displayed calls its own draw() function with the current frame as the argument. | User activates a registered input binding to call display() and update the screen after the action is taken. |
| composeFrame() | Updates the state and coordinates of game objects based on the current state of the object. Called in a tree-like fashion, like display(). | none |
| swapBlocks() | Swaps the blocks that the cursor currently has selected and updates the display. | User activates the registered input binding for swapping blocks. |
| moveCursor() | Moves the cursor to the requested position. The cursor must follow the mouse, but if not using the mouse it will move one block at a time. | User activates the registered input binding for moving blocks. |
| pushRow() | Pushes a new row of blocks onto the Grid. | Called while the Grid is in the Play state, or when the user activates the registered input binding for pushing new rows. When a user pushes a new row, it is pushed faster than the passive speed. |
| pause() | Stop calling composeFrame while the game is paused. Save the current state of the timer, then restore it when it is unpaused so that the clock does not skip ahead. | User activates the input binding registered to pause the game. |
| selectButton() | Calls the action that is registered for an instance of a specific button | User selects a button |

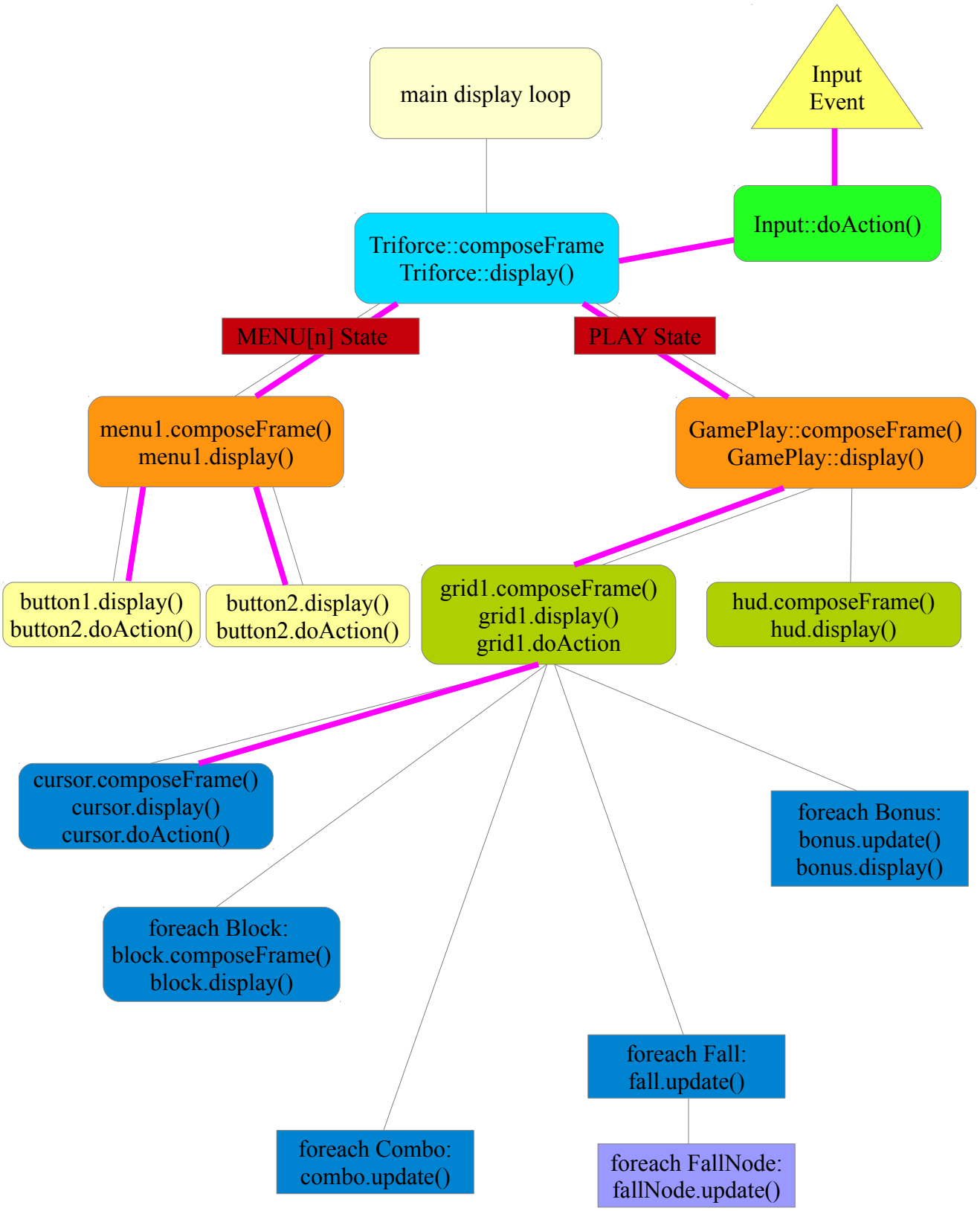
### 3.1.3 Architectural Design

The game's architecure is designed with high cohesion and low coupling in mind. Several classes have been created that abstract their operations. Each object that needs to be displayed has its own display function that is called by its parent class. Likewise, each object that needs a composeFrame has its own as well. Input is initially handled by the Input class, and it is passed from parent to child until it has reached the final callee. The registered keybinding and the state of the game determines which action is taken.

Triforce is highest in the class hierarchy. It passes control between menus and the gameplay state. Menus control buttons. GamePlay controls Grids. A Grid represents a single player and abstracts just about everything so GamePlay does not need to know much about it.

Grid has the most responsibility when it comes to the actual gameplay itself. The objects that Grid controls do not need to interact with many other objects. Grid controls all of the game objects specific to a single player. In addition to calling display(), composeFrame(), and update() for every object that has those functions, it must detect and manage combo, fall, and bonus events. It also must push new rows onto the play area and swap blocks when a user requests it.

While Grid is responsible for controlling everything, it does allow Combo and Fall to make changes to Grid. This is needed so that Grid does not get too bloated. Grid can just pass an instance of itself to Fall and Combo so that they can abstract the operations on them. All Grid has to do is call combo.update() and fall.update() for each combo and fall event, respectively.


main display loop
Input Event
Input::doAction()
Triforce::composeFrame
Triforce::display()
MENU[n] State
PLAY State
menu1.composeFrame()
menu1.display()
GamePlay::composeFrame()
GamePlay::display()
button1.display()
button2.doAction()
button2.display()
button2.doAction()
grid1.composeFrame()
grid1.display()
grid1.doAction
hud.composeFrame()
hud.display()
cursor.composeFrame()
cursor.display()
cursor.doAction()
foreach Bonus:
bonus.update()
bonus.display()
foreach Block:
block.composeFrame()
block.display()
foreach Fall:
fall.update()
foreach Combo:
combo.update()
foreach FallNode:
fallNode.update()

### 3.1.4 Detailed Class Design

I will cover the most complex and vital algorithms, since the game contains too many methods to document them all. The core functions of the game play are to detect combos and falling blocks, as well as

| | |
|---|---|
| Class name | Grid |
| Method name | detectCombo |
| Return type | int |
| Input arguments | Cell & cell, int chains |
| Output arguments | None |
| Error messages | None |
| Files accessed | None |
| Files changed | None |
| Methods invoked | directional match detection methods<br>Combo::Combo<br>Combo::changeState<br>Combo::init |
| Narrative | Method detectCombo takes a pair of coordinates in a Cell class, and the number of chains from the last fall (if any), and detects if a combo occured at that point. |

```
int Grid::detectCombo(Cell &cell, int chains) {
	Combo combo(chains);

	detect left and right matches from given cell (horizontal match)

	if (horizontal match >= minimum combo size) {
		Store left and right coordinates in the combo object

		for each coordinate, horizontally from left to right {
			detect down and up matches (vertical match)

			if (vertical match >= minimum combo size) {
				store up and down coordinates in combo

				set combo state to MULTI
				goto initCombo;
			}
		}

		set combo state to HORIZONTAL
		goto initCombo;
	}
	else {
		detect down and up matches from given cell (vertical match)

		if (vertical match >= minimum combo size) {
			store up and down coordinates in combo

			for each coordinate, vertically from bottom to top {
				detect horizontal matches from each coordinate (horizontal combo)

				if (horizontal match >= minimum combo size) {
					store left and right coordinates in combo

					set combo state to MULTI
					goto initCombo;
				}
			}

			set combo state to VERT
			goto initCombo;
		}
		else {
			set combo state to NONE
			return 0;
		}
	}

	initCombo: // goto label for cleaner code. all valid combos will go here.

	if (combo's chain count >= minimum displayable chain)
		push a new Bonus onto the bonuses list, passing the # of chains and bonus type as CHAIN

	initialize the combo
	push combo back onto list of combo events
	return number of blocks in combo;
}
```

| Class name | Grid |
|---|---|
| Method name | detectFall |
| Return type | bool |
| Input arguments | int row, int col, bool initialize |
| Output arguments | None |
| Error messages | None |
| Files accessed | None |
| Files changed | None |
| Methods invoked | Block::getState<br>Combo::getState<br>Fall::Fall |
| Narrative | Method detectCombo takes a pair of coordinates in a Cell class, and the number of chains from the last fall (if any), and detects if a combo occured at that point. |

```
bool Grid::detectFall(int row, int col, bool initialize) {
        if row > maximum row or row < 0
                return false;

        Block::gameState midState, downState;
        downState = blocks[r-1][c].getState();
        midState = blocks[r][c].getState();

        if (downstate is DISABLED or FALL and midstate is ENABLED) {
                if (initialize) {
                        Fall fall(FallNode(r, c));
                        initialize fall
                        push fall onto fallEvents list
                }
                return true;
        }
        else return false;
}
```

| | |
|---|---|
| Class name | Grid |
| Method name | detectFall |
| Return type | bool |
| Input arguments | Combo & |
| Output arguments | None |
| Error messages | None |
| Files accessed | None |
| Files changed | None |
| Methods invoked | Block::getState<br>Combo::getState<br>Fall::Fall |
| Narrative | Method detectCombo takes a pair of coordinates in a Cell class, and the number of chains from the last fall (if any), and detects if a combo occured at that point. |

```
bool Grid::detectFall(const Combo &combo) {
        Fall fall(combo.chainCount())

        foreach (coordinate in the combo) {
                if (detectFall(row, col, false)) {
                        set the fall type based on the combo type
                        push FallNode onto Fall
                }
        }

        if (there was a fall) {
                fall.possibleChain = true;
                initialize fall
                push fall onto list of Fall events
                return true;
        }
        else return false
}
```

| Class name | Grid |
|---|---|
| Method name | pushRow |
| Return type | void |
| Input arguments | int speed |
| Output arguments | None |
| Error messages | None |
| Files accessed | None |
| Files changed | None |
| Methods invoked | Cursor::offsetY<br>Block::offsetY<br>cursor::shiftRow<br>Grid::addRow |
| Narrative | Method pushRow updates the position of the blocks when a row needs to be pushed onto the gameplay area. The speed argument tells how fast it will push. The speed that rows are pushed gradually accels over time. The speed that rows are pushed when a user presses the push row keybinding is fixed. |

```
void Grid::pushRow(int speed) {
        pushOffset += speed;

        cursor->offsetY(-speed);
        foreach Block in the Grid
                blocks[i][j].offsetY(-speed);

        if (pushOffset >= blockLength) {
                pushOffset %= blockLength;
                cursor.shiftRow();
                addRow();
        }
}
```

## 3.2 Walkthrough

Spellcheck
Draw arrows for your architectural design to indicate flow
Reevaluate CRC cards; update for latest design
Do architectural design according to standards
Update PDL's and tabular design diagrams to latest design

- Brandon Hinesley

# 3.3 Revised Design after Walkthrough

## 3.1.1 Component Design

### CRC Cards

| **Class: Game/Triforce** |
| --- |
| **Responsibility:**<br>Load the main menu<br>Control flow of the program based on the current state<br>Set up input bindings/handlers |
| **Collaboration:**<br>GamePlay<br>Input |

| **Class: Input** |
| --- |
| **Responsibility:**<br>Provide an interface to bind input events to actions (callback methods) |
| **Collaboration:**<br>Triforce<br>GamePlay<br>Button<br>Grid<br>Cursor |

| **Class: GamePlay** |
| --- |
| **Responsibility:**<br>Initialize the game<br>Handle input that affects the state of GamePlay<br>Control the flow among objects in GamePlay<br>Initialize objects that will be displayed during the game |
| **Collaboration:**<br>Grid<br>Input<br>HUD |

**Class: Grid**

**Responsibility:**
Provide an interface to GamePlay for control the cursor
Update the Cursor's coordinates
Swap Blocks
Detect/Initialize Combos
Detect/Initialize Falling Events
Initialize Bonuses
Keep track of all events and update them for each iteration of the main loop
Call display methods for all of Grid's objects that must be displayed
Update score

**Collaboration:**
Cursor
Block
Combo
Fall
Bonus
Input

**Class: Cursor**

**Responsibility:**
Draw itself
Provide methods to move the cursor
Keep track of position

**Collaboration:**
Input
Grid

**Class: Block**

**Responsibility:**
Display itself
Provide methods for swapping Blocks
Provide methods for falling down

**Collaboration:**
Grid
Combo
FallNode

**Class: Combo**

**Responsibility:**
Provide an interface to Grid for managing combo events
Update the state of the Combo
Update the Grid's state based on the state of the Combo
Keep track of chains
Detect fall events when the combo has elapsed

**Collaboration:**
Grid
Block
Fall

**Class: Fall**

**Responsibility:**
Provide an interface to Grid for managing fall events
Update the state of the fall
Manage individual FallNodes, calling their update functions for every iteration of a Fall
Keep track of chains
Detect combo events when the fall has elapsed

**Collaboration:**
Grid
Block
Combo
FallNode

**Class: FallNode**

**Responsibility:**
Provide an interface to Fall for managing a single list column of falling blocks
Update state of FallNode
Keep track of position and number of Blocks in the fall
Cause Blocks to fall for each iteration

**Collaboration:**
Fall
Grid
Block

**Class: Button**

**Responsibility:**
Display itself based on current state
Register and call callback functions when pressed

**Collaboration:**
Game
GamePlay
Action

### 3.3.2 Interface Design

The currently displayed objects are controlled by the state of the game. States are defined as enums that contain all possible states. Different functions get called based on the state of the game. For example, when the game is in the Play state, the GamePlay's display() and composeFrame() are called, which call Grid's respective functions. Grid calls Block, Cursor, and Bonus display() and composeFrame() functions.

| Interface Actions | Interface Objects | User Actions (events) |
|---|---|---|
| display() | Calls display() methods for each class, starting at the base and branching out in a tree fashion. Every object that is actually displayed calls its own draw() function with the current frame as the argument. | User activates a registered input binding to call display() and update the screen after the action is taken. |
| composeFrame() | Updates the state and coordinates of game objects based on the current state of the object. Called in a tree-like fashion, like display(). | none |
| swapBlocks() | Swaps the blocks that the cursor currently has selected and updates the display. | User activates the registered input binding for swapping blocks. |
| moveCursor() | Moves the cursor to the requested position. The cursor must follow the mouse, but if not using the mouse it will move one block at a time. | User activates the registered input binding for moving blocks. |
| pushRow() | Pushes a new row of blocks onto the Grid. | Called while the Grid is in the Play state, or when the user activates the registered input binding for pushing new rows. When a user pushes a new row, it is pushed faster than the passive speed. |
| pause() | Stop calling composeFrame while the game is paused. Save the current state of the timer, then restore it when it is unpaused so that the clock does not skip ahead. | User activates the input binding registered to pause the game. |
| selectButton() | Calls the action that is registered for an instance of a specific button | User selects a button |

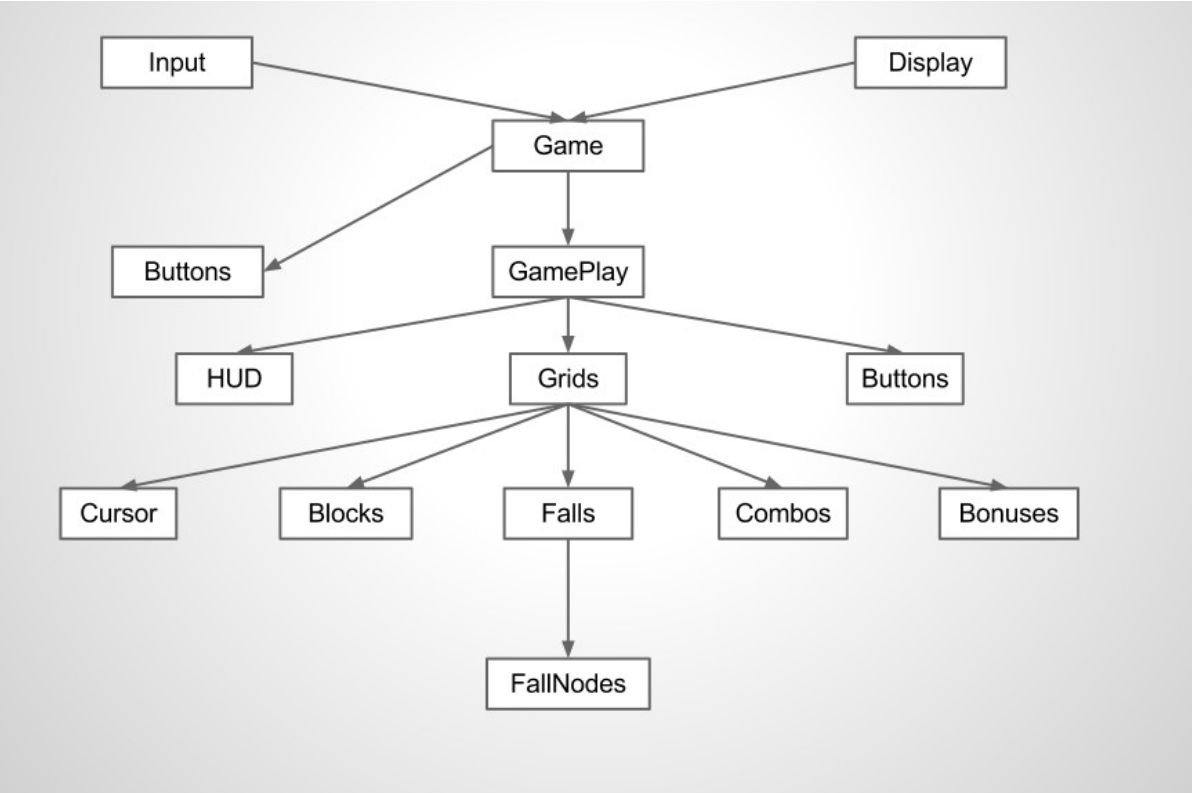
### 3.3.3 Architectural Design

The game's architecure is designed with high cohesion and low coupling in mind. Several classes have been created that abstract their operations. Each object that needs to be displayed has its own display function that is called by its parent class. Likewise, each object that needs a composeFrame has its own as well. Input is initially handled by the Input class, and it is passed from parent to child until it has reached the final callee. The registered keybinding and the state of the game determines which action is taken.

Triforce is highest in the class hierarchy. It passes control between menus and the gameplay state. Menus control buttons. GamePlay controls Grids. A Grid represents a single player and abstracts just about everything so GamePlay does not need to know much about it.

Grid has the most responsibility when it comes to the actual gameplay itself. The objects that Grid controls do not need to interact with many other objects. Grid controls all of the game objects specific to a single player. In addition to calling display(), composeFrame(), and update() for every object that has those functions, it must detect and manage combo, fall, and bonus events. It also must push new rows onto the play area and swap blocks when a user requests it.

While Grid is responsible for controlling everything, it does allow Combo and Fall to make changes to Grid. This is needed so that Grid does not get too bloated. Grid can just pass an instance of itself to Fall and Combo so that they can abstract the operations on them. All Grid has to do is call combo.update() and fall.update() for each combo and fall event, respectively.




*Architectural Design Diagram*

## 3.3.4 Detailed Class Design

I will cover the most complex and vital algorithms, since the game contains too many methods to document them all. The core functions of the game play are to detect combos and falling blocks, as well as

| | |
|---|---|
| Class name | Grid |
| Method name | detectCombo |
| Return type | int |
| Input arguments | Cell & cell, int chains, bool doBonus |
| Output arguments | None |
| Error messages | None |
| Files accessed | None |
| Files changed | None |
| Methods invoked | Grid:: matchUp, matchDown, matchLeft, matchRight<br>Combo::changeState<br>Combo::init<br>Grid::initBonus |
| Narrative | Method detectCombo takes a pair of coordinates in a Cell class, and the number of chains from the last fall (if any), and detects if a combo occured at that point. If doBonus is true, it will spawn a bonus animation if a big combo or chain is detected. |

```
int Grid::detectCombo(Cell &cell, int chains, bool doBonus) {
	Combo combo(chains);

	detect left and right matches from given cell (horizontal match)

	if (horizontal match >= minimum combo size) {
		Store left and right coordinates in the combo object

		for each coordinate, horizontally from left to right {
			detect down and up matches (vertical match)

			if (vertical match >= minimum combo size) {
				store up and down coordinates in combo

				set combo state to MULTI
				initialize combo
				if (doBonus)
					initialize bonus
			}
		}

		set combo state to HORIZONTAL
		initialize combo
		if (doBonus)
			initialize bonus
		return combo
	}
	else {
		detect down and up matches from given cell (vertical match)

		if (vertical match >= minimum combo size) {
			store up and down coordinates in combo

			for each coordinate, vertically from bottom to top {
				detect horizontal matches from each coordinate (horizontal combo)

				if (horizontal match >= minimum combo size) {
					store left and right coordinates in combo

					set combo state to MULTI
					initialize combo
					if (doBonus)
						initialize bonus
				}
			}
			set combo state to VERT
			initialize combo
			if (doBonus)  initialize bonus
		}
		else {
			set combo state to NONE
			return combo;
		}
	}
}
```

| Class name | Grid |
|---|---|
| Method name | detectFall |
| Return type | bool |
| Input arguments | int row, int col, bool initialize |
| Output arguments | None |
| Error messages | None |
| Files accessed | None |
| Files changed | None |
| Methods invoked | Block::getState<br>Combo::getState<br>Fall::Fall |
| Narrative | Method detectFall detects a fall from a given set a coordinates. It will initialize the fall as well if *initialize* is true. |

```
bool Grid::detectFall(int row, int col, bool initialize) {
        if row > maximum row or row < 0
                return false;

        Block::gameState midState, downState;
        downState = blocks[r-1][c].getState();
        midState = blocks[r][c].getState();

        if (downstate is DISABLED or FALL and midstate is ENABLED) {
                if (initialize) {
                        Fall fall(FallNode(r, c));
                        initialize fall
                        push fall onto fallEvents list
                }
                return true;
        }
        else return false;
}
```

| | |
|---|---|
| Class name | Grid |
| Method name | detectFall |
| Return type | bool |
| Input arguments | Combo & |
| Output arguments | None |
| Error messages | None |
| Files accessed | None |
| Files changed | None |
| Methods invoked | Block::getState<br>Combo::getState<br>Fall::Fall |
| Narrative | Method detectFall detects a fall for all blocks in a given combo. |

```
bool Grid::detectFall(const Combo &combo) {
        Fall fall(combo.chainCount())

        foreach (coordinate in the combo) {
                if (detectFall(row, col, false)) {
                        set the fall type based on the combo type
                        push FallNode onto Fall
                }
        }

        if (there was a fall) {
                fall.possibleChain = true;
                initialize fall
                push fall onto list of Fall events
                return true;
        }
        else return false
}
```

| Class name | Grid |
|---|---|
| Method name | pushRow |
| Return type | void |
| Input arguments | None |
| Output arguments | None |
| Error messages | None |
| Files accessed | None |
| Files changed | None |
| Methods invoked | Cursor::offsetY<br>Block::offsetY<br>cursor::shiftRow<br>Grid::addRow |
| Narrative | Method pushRow updates the position of the blocks when a row needs to be pushed onto the gameplay area. The interval that rows push up decreases over time, but that is not controlled by composeFrame. The interval for force-pushing rows up is fixed. |

```
void Grid::pushRow() {
	pushOffset += speed;

	cursor->offsetY(-speed);
	foreach Block in the Grid
		blocks[i][j].offsetY(-speed);

	if (pushOffset >= blockLength) {
		pushOffset %= blockLength;
		cursor.shiftRow();
		addRow();
	}
}
```

# Section 4: Conclusion

## 4.1 Conclusion

Overall, I found this project to be a good learning experience. I love to program, so my gut instinct was to start programming without caring too much about the documentation. However, I have learned the importance of documentation. I had to go back and fix my design several times, which would have been a very expensive mistake had it been in a business setting. Although I am happy with the outcome of the game, I think more time spent on design would have made it so I didn't have to refactor so much. In the future, I will draw out my ideas with diagrams and discuss them with my teammates more before implementing them.