<^> TRIFORCE <^>

Final Document

By: Brandon Hinesley
CMPS 335, Arif Wani
Other Team member: Gabe Pike

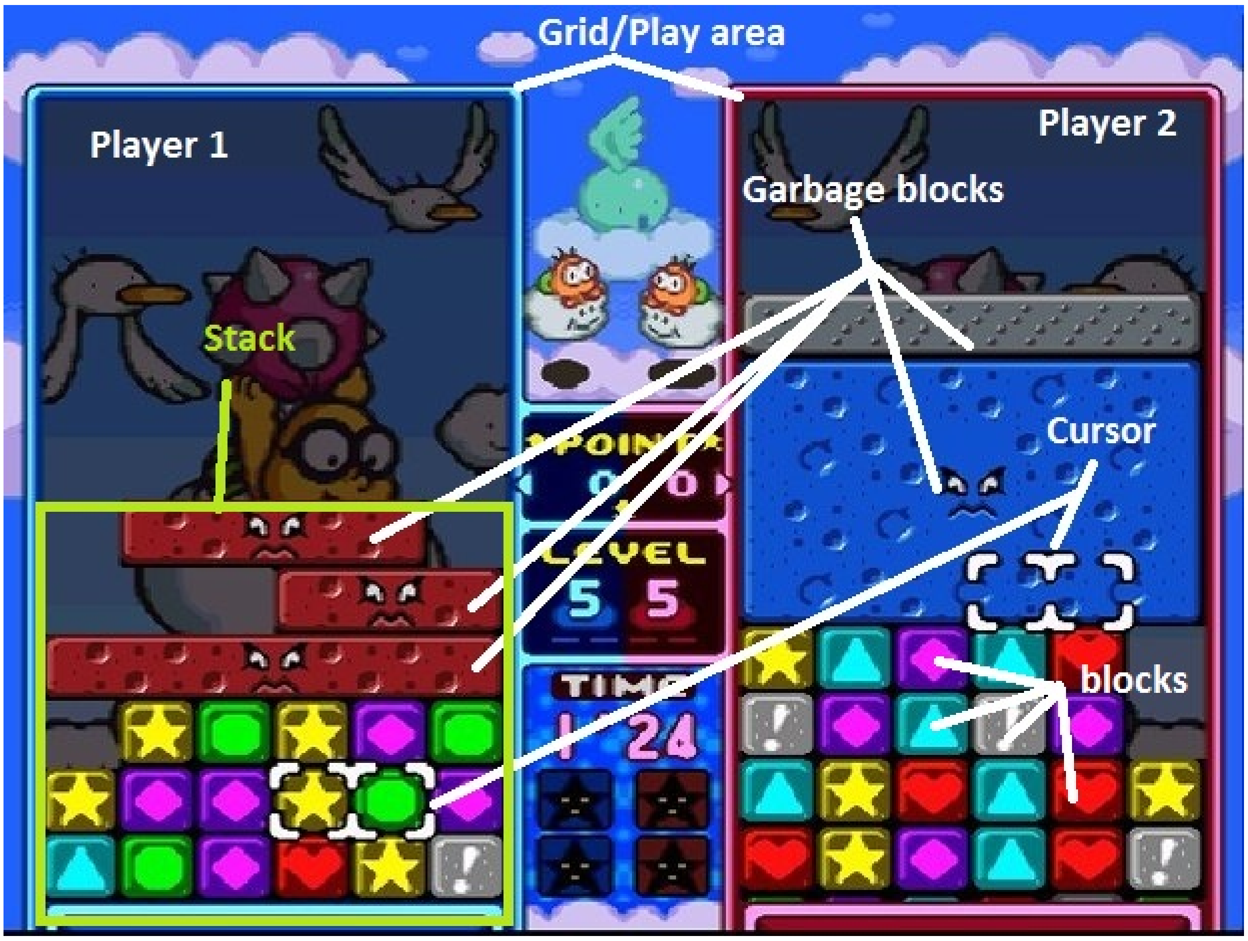
Table of Contents

Section 1: Requirements workflow

1.1 Requirements workflow

1.1.1 Initial understanding of the project

Triforce is a clone of the original Tetris Attack (a Super Nintendo game), with modern graphics and features.



Tetris Attack in 2-player mode

There are single and 2-player modes. The object of the 2-player game is to cause your human opponent's blocks to reach the top of the grid, at which point they lose. The object of the single player game is to prevent your blocks from reaching the top of the grid while the game speeds up. In all game modes, the normal blocks rise from the bottom, and the garbage blocks fall from the top. There are two principal operations in the game: raising the stack of blocks manually (more quickly), and swapping the two blocks under the cursor. By swapping blocks, they can be rearranged to form combinations, which cause the blocks to clear. Combinations consist of 3 or more blocks of the same type directly adjacent to each other in a row or column.

Chains are the most critical (and difficult) aspect of the game. Garbage blocks can be dropped on your opponent by forming chains, which are multiple combinations occuring in a related sequence. The greater the chain multiplier, the larger the garbage block.

An interesting aspect of the game, is that the rising of blocks from below pauses under certain conditions. This allows players to respond to difficult situations (or be rewarded for clearing blocks), making the game less frustrating than it might otherwise be. It also allows for comebacks from seemingly impossible situations.

Glossary

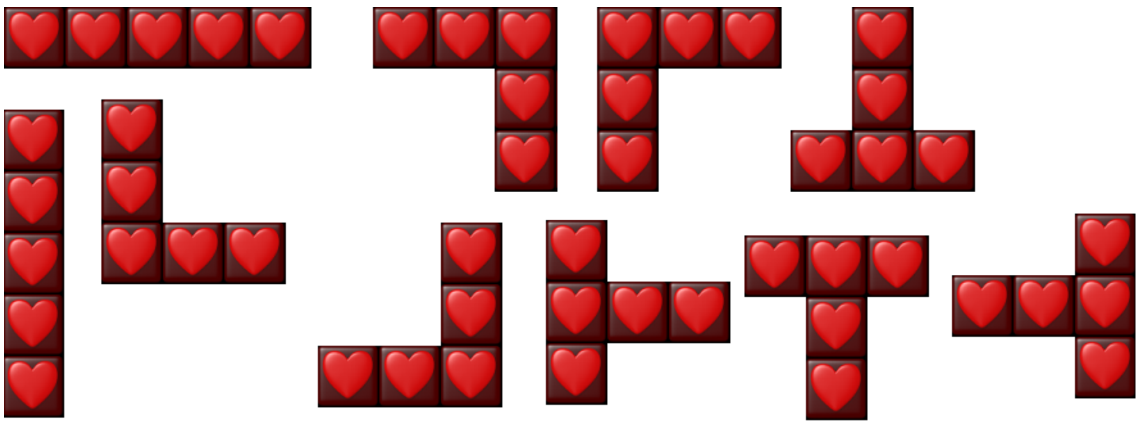
- **Clearing blocks (aka breaking)** – Blocks are cleared whenever a combo occurs. They dissapear, and any blocks above them fall.
- **Clearing garbage blocks -** Breaking garbage blocks *(see Garbage blocks)* is somewhat different. They are indirectly cleared by a combination of normal blocks occuring directly adjacent to them. When a garbage block is cleared, the first (and possibly only) row of the garbage block changes into normal blocks. Any combinations made as these normal blocks fall count towards the chain multiplier *(see Chains)*. If any garbage blocks of the same color are adjacent to a cleared garbage block, they are also cleared in the same fashion.
- **Combination (aka combo)** – Combinations are one of the most basic elements in the game. They occur when 3 or more blocks are in a row, at which point all blocks used to create the combination are cleared. The greater then number of blocks in the combination, the greater the score for clearing them. Note that in some cases, blocks of different types can be used to create a single large combination.
    - **3x Combo**
        - smallest possible combo
        - 3 of a given block type A in one of two possible ways

Every possible 3x combo

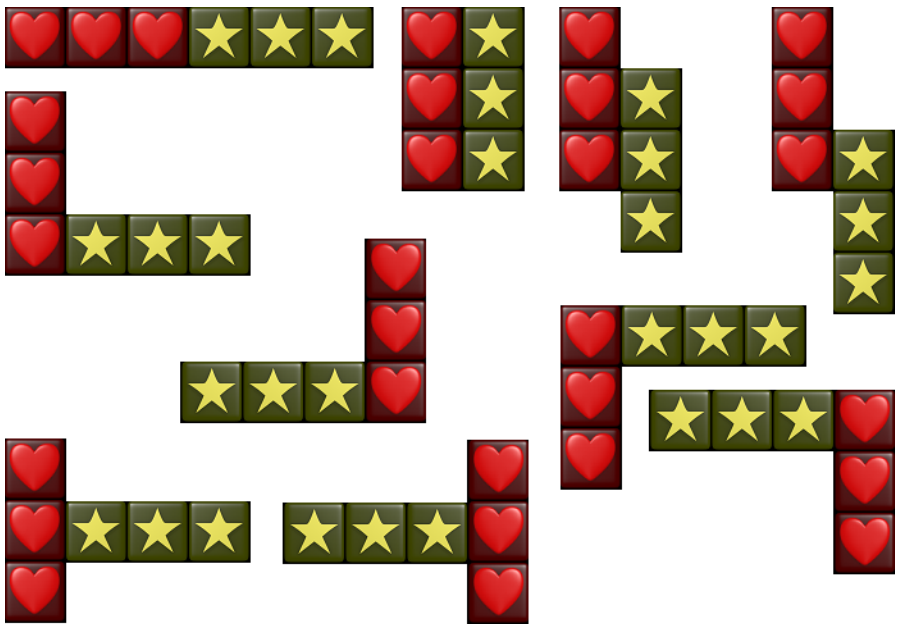
    - **4x Combo**
        - 4 of a given block type A in one of two possible ways

Every possible 4x combo

    - **5x Combo**
        - 5 of a given block type A in one of 10 possible ways


Every possible 5x combo

- **6x Combo**
    - 3 of a given block type A and 3 of a given block type B (or A) in one of many possible ways


Every possible 6x combo

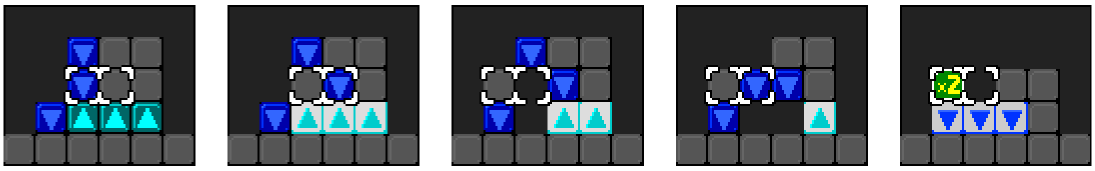
    -

- **7x Combo**
    - 4 of a block type A and 3 of a block type B (or A) in one of many possible ways
- **8x Combo**
    - 5 of a block type A and 3 of a block type B (or A) or
    - 4 of a block type A and 4 of a block type B (or A)
- **9x Combo**
    - 5 of a block type A and 4 of a block type B (or A)
- **10x Combo**
    - largest possible combo

        - 5 of a block type A and 5 of a block type B (or A)

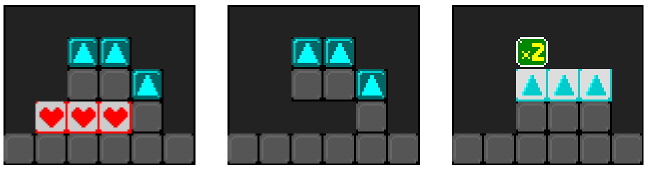
- **Chains** – Chains of combinations occur when more than one combination occurs as a direct result of blocks falling due to a prior combination. A (theoretically) unlimited length of chains are possible. Chains act as multipliers to the score that would normally be calculated for a combination. Chaining together N amount of combinations result in a score multiple for that run of xN. From the original Tetris Attack, here are the 5 different ways that chains can occur, in order of increasing difficulty:

    - **Basic Chain**



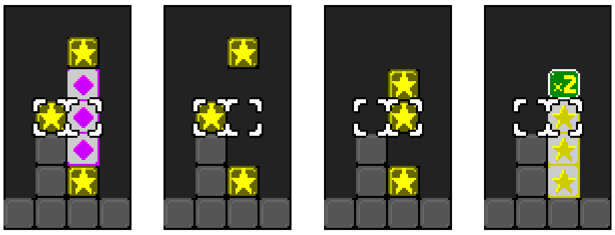
Basic chain

    - **Raised Chain**
        - the first combination is not adjacent to the chained combination



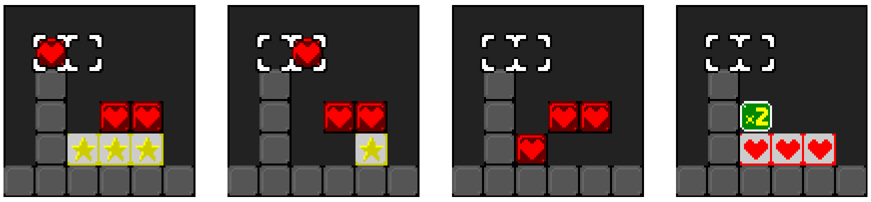
Raised chain

    - **Skill Chain**
        - a chain quickly made by the player by moving a block into the space left from the first combination, creating a new combination once the blocks fall



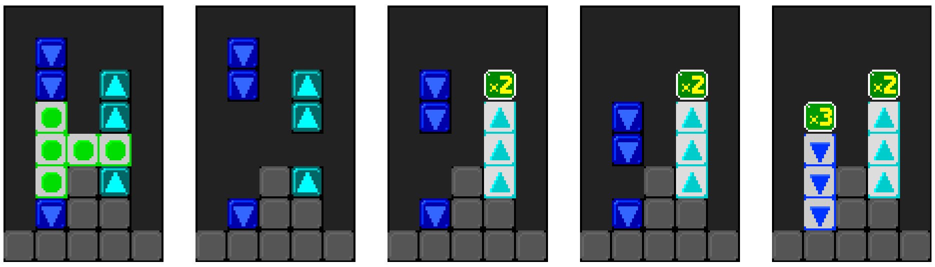
    - *Skill chain*

    - **Advanced Chain**
        - these chains take advantage of the fact that there is a split second before a combination is cleared, where the blocks directly above it have not yet fallen, but it is possible for the player to drop another block



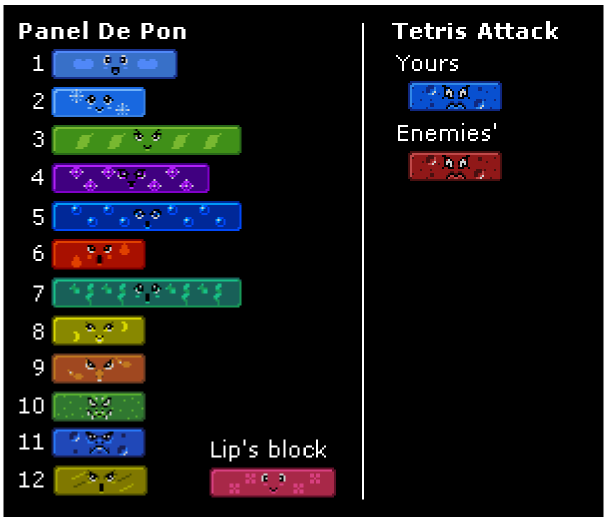
Advanced chain

- **Time-Lag Chain**
  - involve 2 or more combinations being cleared as a direct result of the first combination being cleared



Time-lag chain

- **Garbage blocks** – Special blocks of varying width, which turn into normal blocks when cleared *(see Clearing garbage blocks)*. The sole purpose of garbage blocks is to block players from making combinations, while simultaneously increasing the height of their stack. Garbage blocks fall at random in single player mode. In 2-player mode, garbage blocks fall as a result of chains made by opposing players.



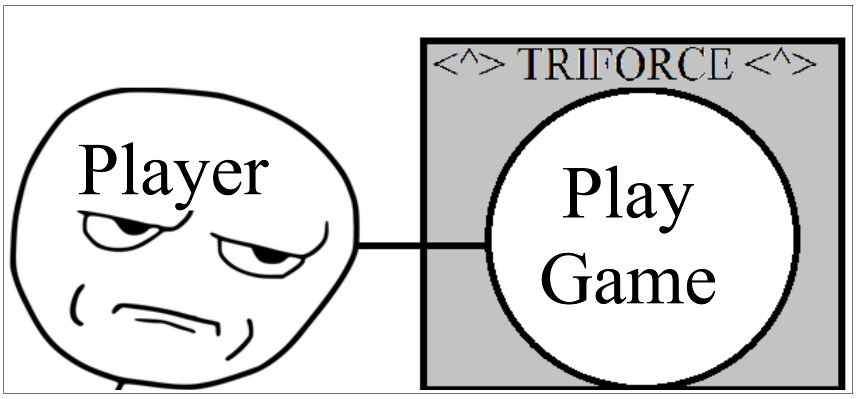
Garbage blocks from Panel De Pon and Tetris Attack

- **Raise blocks** – The blocks naturally rise as time progresses. The player can manually speed up this process if they want more blocks on the screen. It is possible to raise them past the top of the grid, and lose the game.
- **Stack (aka Grid** – The stack consists of the play area, which holds blocks and the cursor.
- **Swapping blocks** – The only operation that can be directly performed on blocks. The swap in the left part of the cursor swaps places with the block in the right part of the cursor. Other than moving the cursor and raising the stack *(see Raising the stack)*, there is no other input action that can be performed in-game.

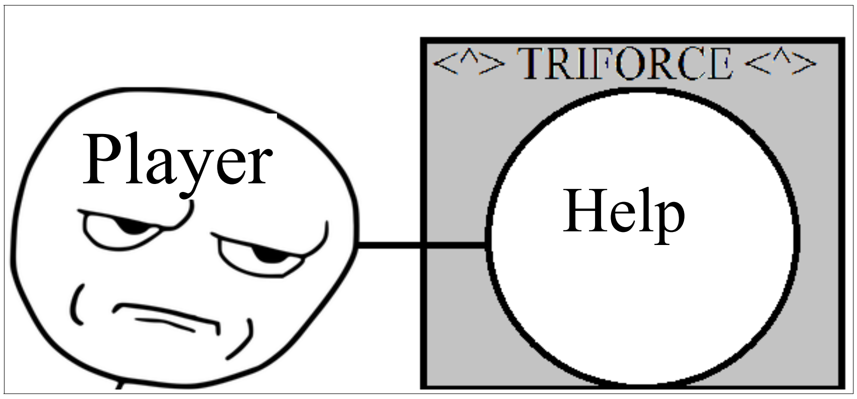
1.1.2 Initial business model

Initial processes:

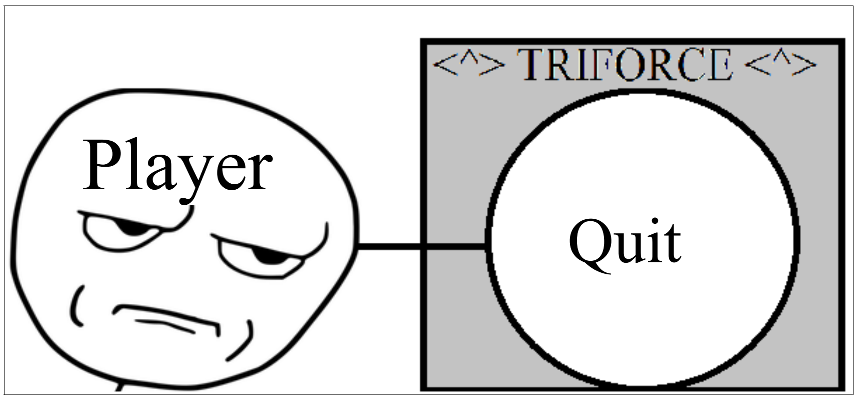
- Play game
    - Control cursor
    - Speed up blocks
    - Swap blocks
- Help
- Quit



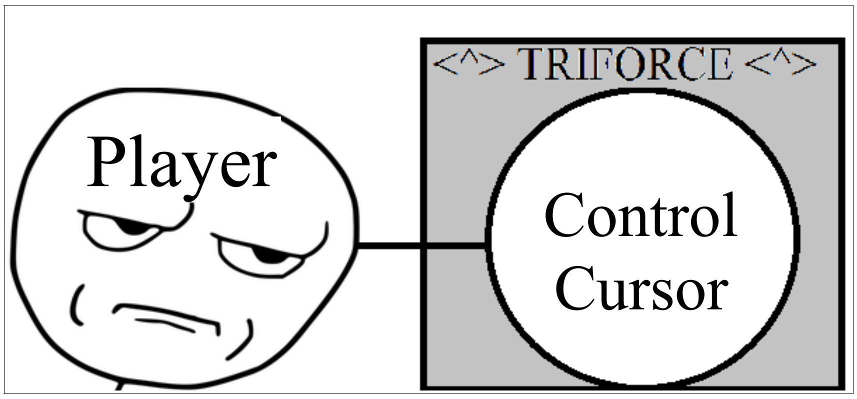
Brief Description: When the Player activates the Play Game menu item, the game state changes to Play, and the game starts.



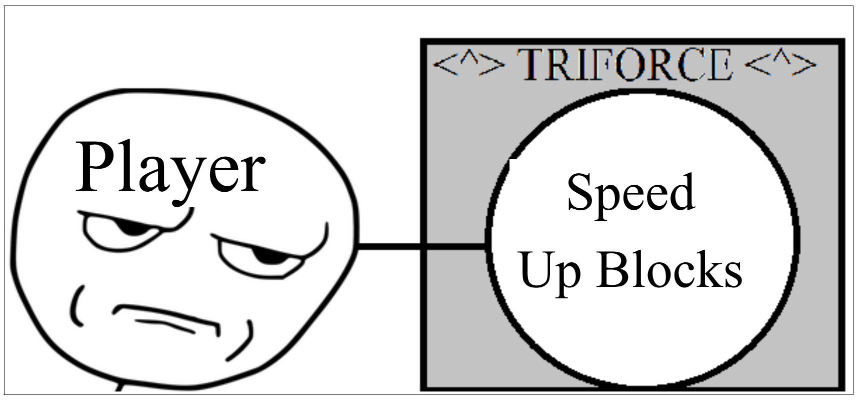
Brief Description: When the Player activates the Help menu item, the game displays instructions of the various actions that are possible in the game, and graphics of how different combinations/chains are made.



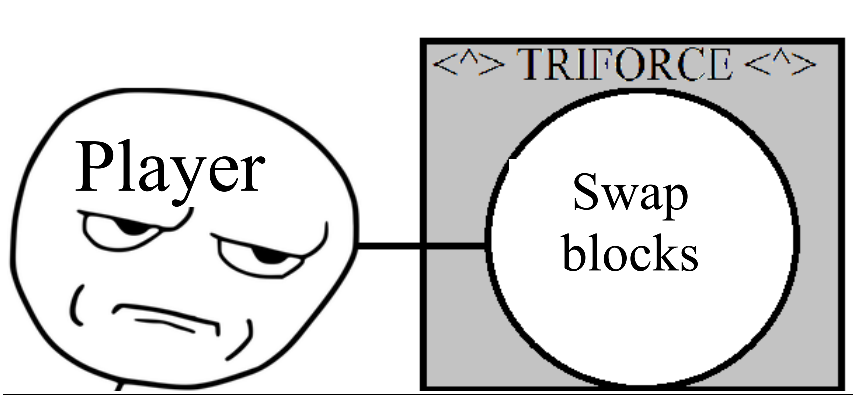
Brief Description: When the Player activates the Quit menu item, the game asks the user if they are sure, and then quits upon 'yes', or goes back to the menu upon 'no'.



Brief Description: In the Play state, the user has control of a special in-game cursor for selecting two blocks at a time. The cursor can move up, down, left, and right, but cannot move backward or foreward, rotate clockwise or counterclockwise, or derotate.

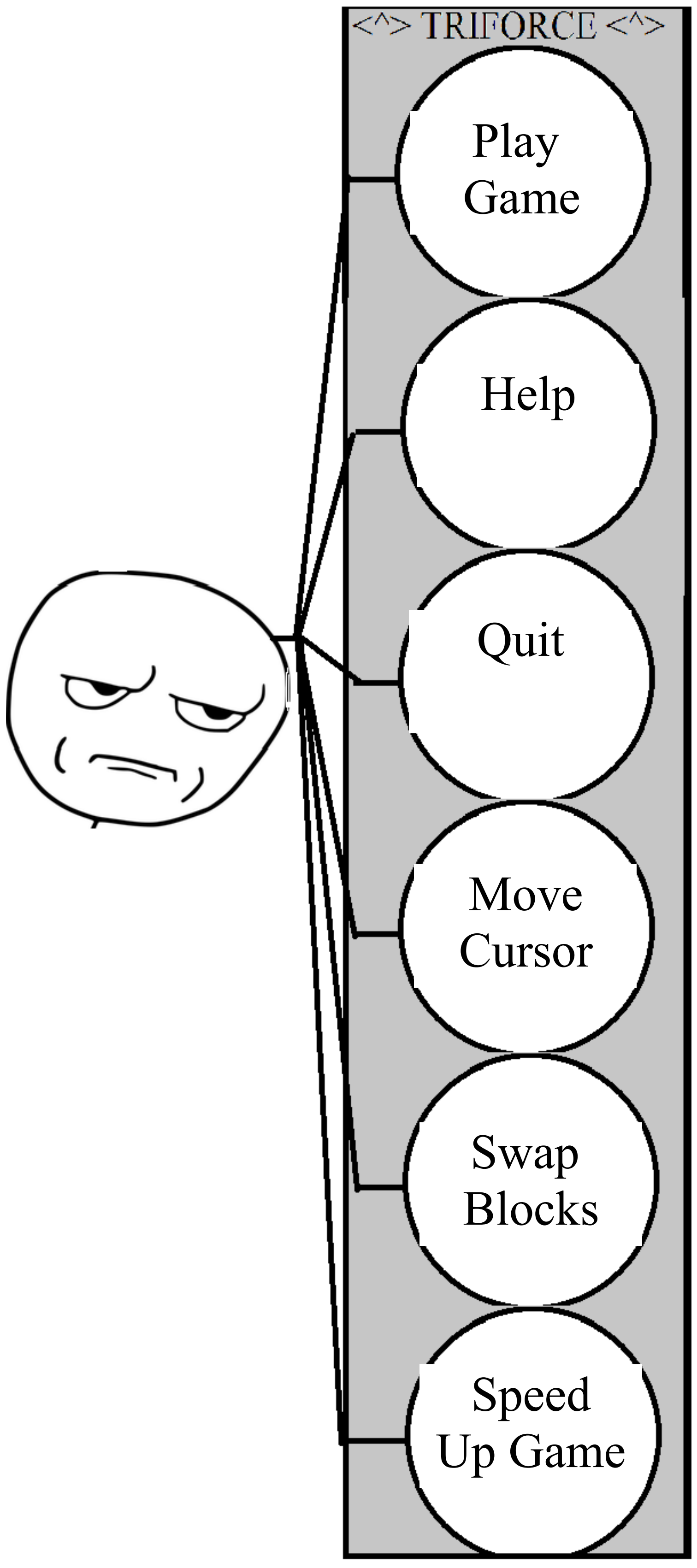


Brief Description: In the Play state, when the Player activates the Speed Up Blocks function, the Grid will speed up its normal action of moving the blocks up.



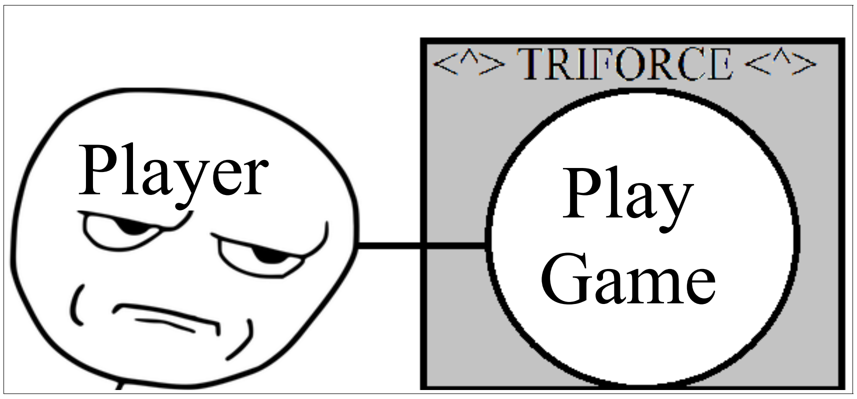
Brief Description: When the Player activates the Swap Blocks function, the cursor swaps the two blocks that are underneath it.

Summary:



<^> TRIFORCE <^>
Play Game
Help
Quit
Move Cursor
Swap Blocks
Speed Up Game

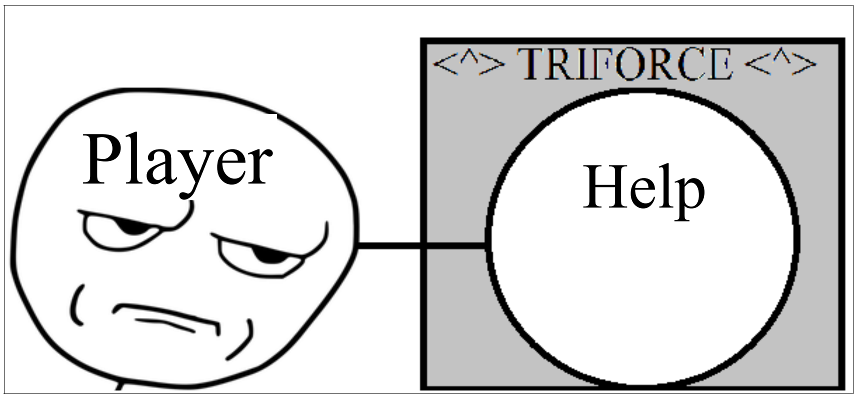
1.1.3 Initial requirements



Brief Description: When the Player activates the Play Game menu item, the game state changes to Play, and the game starts.

Step-by-Step Description:

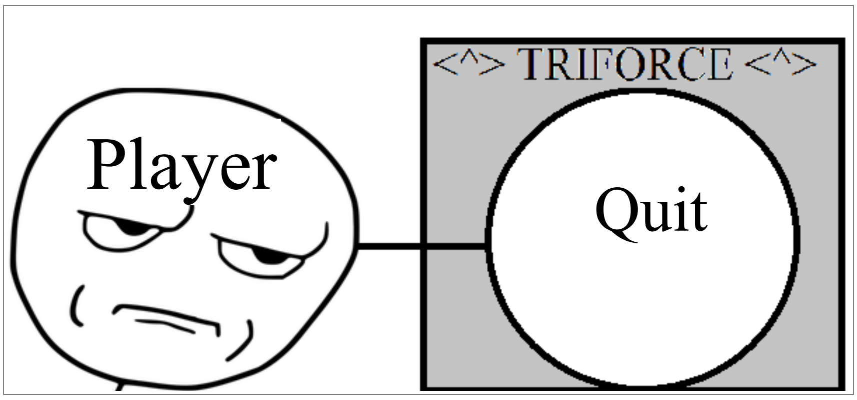
1. Load Grid, Cursor, and blocks
2. Start game



Brief Description: When the Player activates the Help menu item, the game displays instructions of the various actions that are possible in the game, and graphics of how different combinations/chains are made.

Step-by-Step Description:

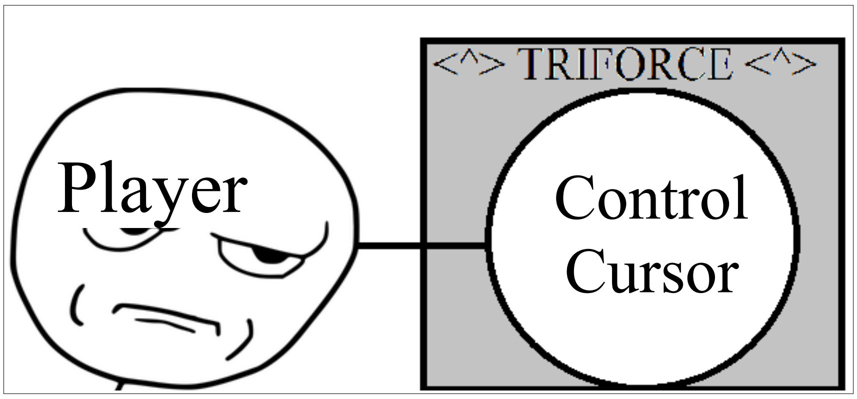
1. Load and display Help information
2. Return to menu



Brief Description: When the Player activates the Quit menu item, the game asks the user if they are sure, and then quits upon 'yes', or goes back to the menu upon 'no'.

Step-by-Step Description:

1. Display dialog asking user to confirm that they would like to quit
2. Quit or return to menu

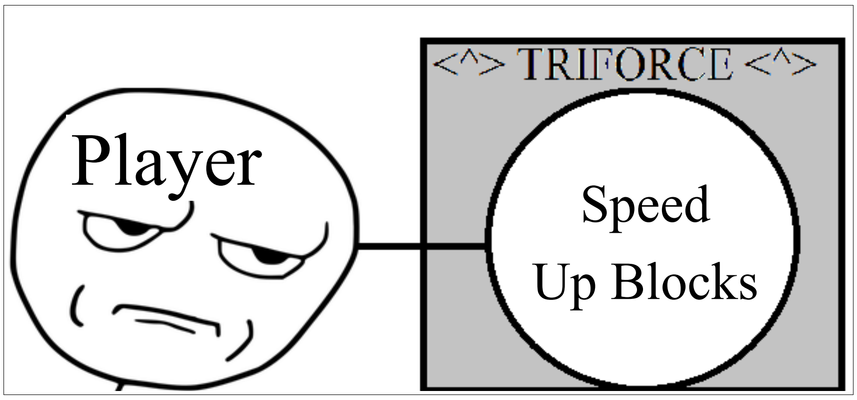


Brief Description: In the Play state, the user has control of a special in-game cursor for selecting two blocks at a time. The cursor can move up, down, left, and right. When the Player activates the Swap Blocks function, the cursor swaps the two blocks that are underneath it.

Step-by-Step Description:

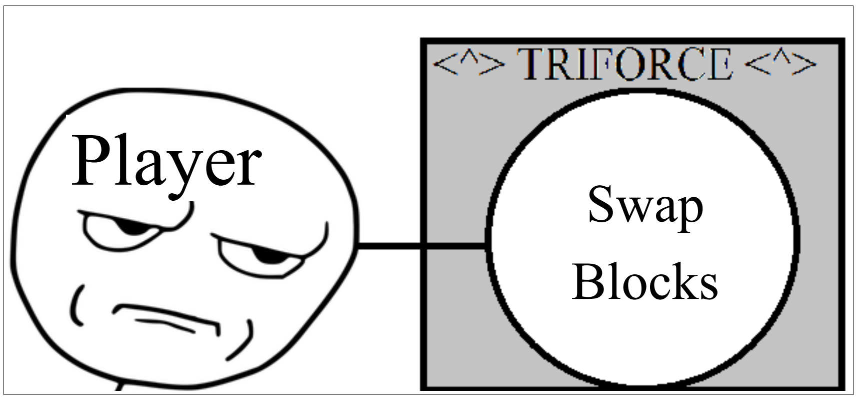
1. For each input action, attempt to move the Cursor accordingly.
2. Move in the direction the user requested.
3. Clear blocks if a combination was made in the last step.
4. Add points to score if the current chain has ended.
5. Go back to step 1.

Brief Description: In the Play state, when the Player activates the Speed Up Blocks function, the Grid will speed up it's normal action of moving the blocks up.



Step-by-Step Description:

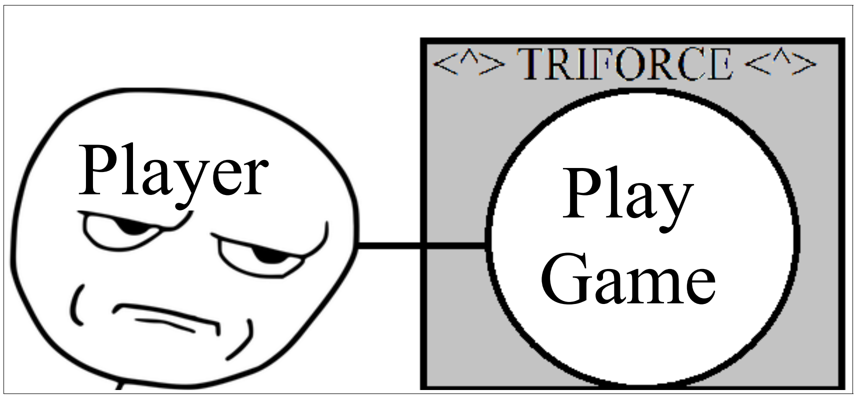
1. Move the stack up a larger amount than normal
2. If the Speed Up Blocks button is still being pressed, goto 1, otherwise continue raising stack at the normal rate.



Step-by-Step Description:

1. Exchange the block in the right side of the cursor with the block in the left side.

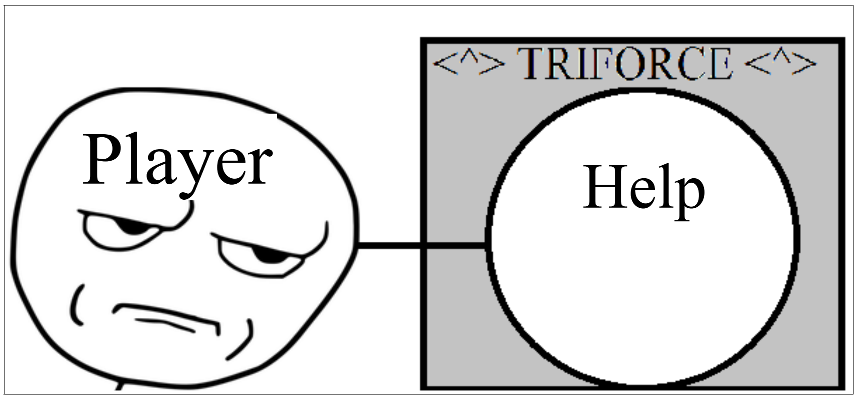
1.1.5 Iterated use-case diagrams and updated descriptions



Brief Description: When the Player activates the Play Game menu item, the game state changes to Play, and the game starts.

Step-by-Step Description:

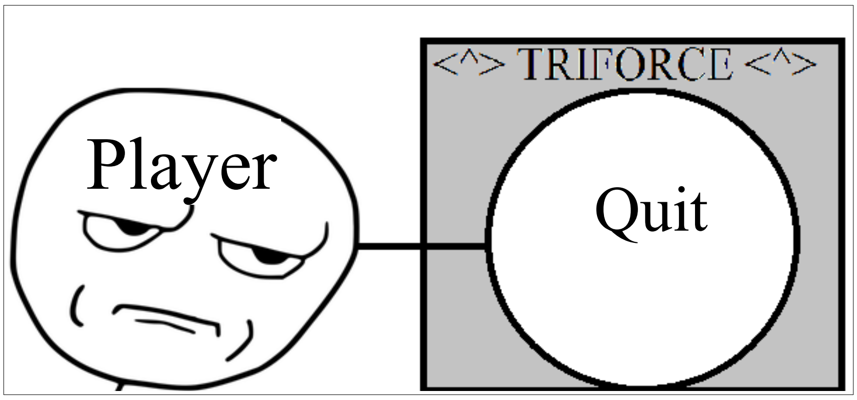
3. Load Grid, Cursor, and blocks
4. Countdown 3... 2... 1...
5. Start game



Brief Description: When the Player activates the Help menu item, the game displays instructions of the various actions that are possible in the game, and graphics of how different combinations/chains are made.

Step-by-Step Description:

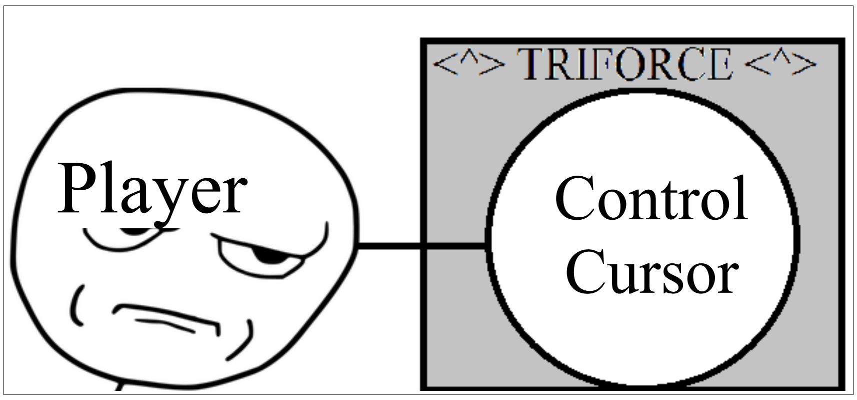
1. Load and display Help information, which is hard-coded text and images
2. Return to menu



Brief Description: When the Player activates the Quit menu item, the game asks the user if they are sure, and then quits upon 'yes', or goes back to the menu upon 'no'.

Step-by-Step Description:

1. Display dialog asking user to confirm that they would like to quit
2. Quit (if user selected yes) or return to menu (if user selected no)

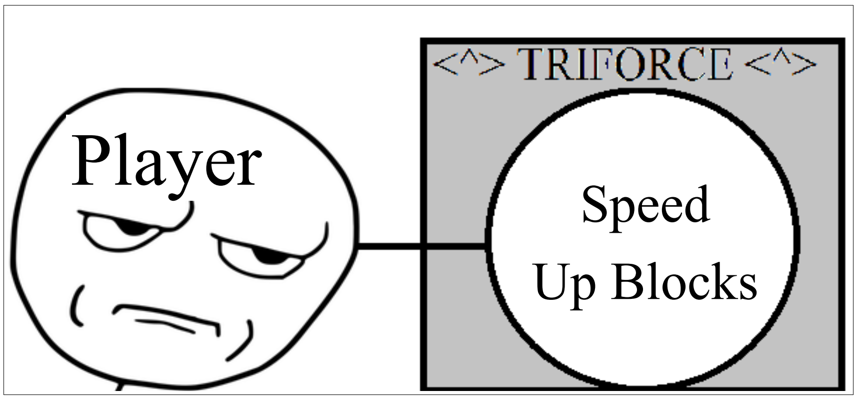


Brief Description: In the Play state, the user has control of a special in-game cursor for selecting two blocks at a time. The cursor can move up, down, left, and right. When the Player activates the Swap Blocks function, the cursor swaps the two blocks that are underneath it.

Step-by-Step Description:

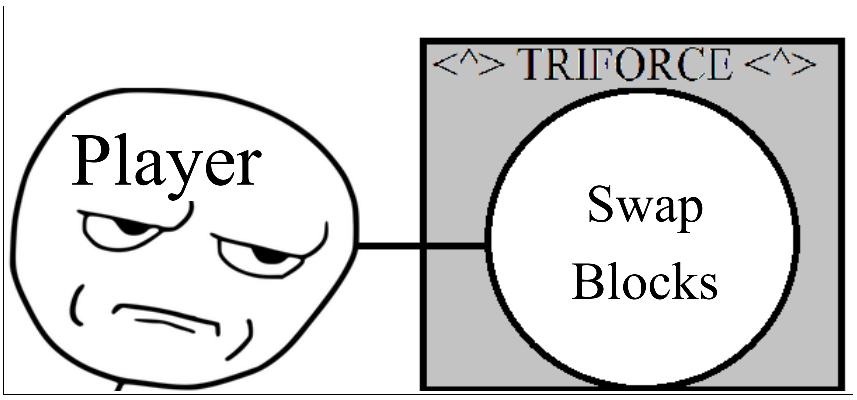
1. For each input action, attempt to move the Cursor accordingly.
2. If the user pressed Up, go up unless the Cursor is at the top of the grid already. If the user pressed Down, go down unless the block below the current block is not enabled yet (i.e. It's still being pushed onto the grid). If the user pressed left or right, move the Cursor left or right, unless the Cursor is already on the edge of the grid. If the user pressed swap, then swap the blocks selected by the cursor.
3. Clear blocks if a combination was made in the last step.
4. Add points to score if the current chain has ended.
5. Go back to step 1.

Brief Description: In the Play state, when the Player activates the Speed Up Blocks function, the Grid will speed up it's normal action of moving the blocks up.



Step-by-Step Description:

1. Move the stack up a larger amount than normal
2. If the Speed Up Blocks button is still being pressed, goto 1, otherwise continue raising stack at the normal rate.



Step-by-Step Description:

1. Exchange the block in the right side of the cursor with the block in the left side.

1.2 Walkthrough

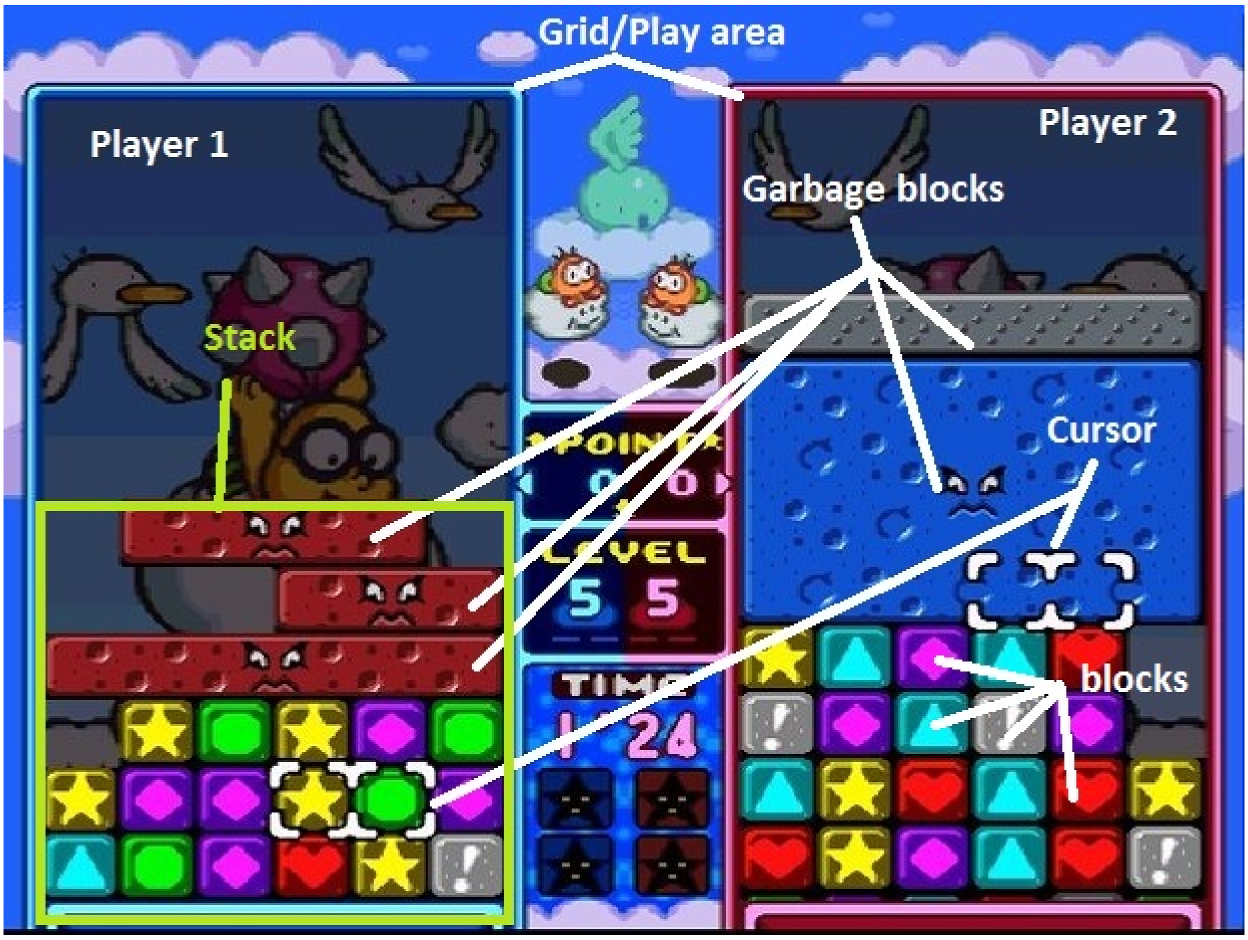
- By Gabe Pike

Explain how input will be handled
Explain how the menu system will be
Realistically assess which goals can be achieved within the time frame (especially consider 2-Player mode, proper chaining, and garbage blocks)
What tools, libraries, and techniques will be required?

1.3 Revised Requirements after walkthrough

Revised Understanding of the project

Triforce is a clone of the original Tetris Attack (a Super Nintendo game), with modern graphics and features.



Tetris Attack in 2-player mode

There are single and 2-player modes. The object of the 2-player game is to cause your human opponent's blocks to reach the top of the grid, at which point they lose. The object of the single player game is to prevent your blocks from reaching the top of the grid while the game speeds up. In all game modes, the normal blocks rise from the bottom, and the garbage blocks fall from the top. There are two principal operations in the game: raising the stack of blocks manually (more quickly), and swapping the two blocks under the cursor. By swapping blocks, they can be rearranged to form combinations, which cause the blocks to clear. Combinations consist of 3 or more blocks of the same type directly adjacent to each other in a row or column.

Chains are the most critical (and difficult) aspect of the game. Garbage blocks can be dropped on your opponent by forming chains, which are multiple combinations occuring in a related sequence. The greater the chain multiplier, the larger the garbage block.

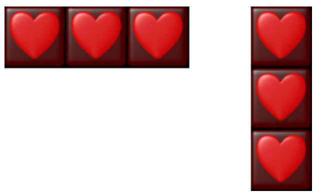
An interesting aspect of the game, is that the rising of blocks from below pauses under certain conditions. This allows players to respond to difficult situations (or be rewarded for clearing blocks), making the game less frustrating than it might otherwise be. It also allows for comebacks from seemingly impossible situations.

We will focus on the single player mode and player vs. player aspects of the game. Single player will be implemented in an endurance mode, not including player vs. AI. While no networking support will be added, the game will be designed in such a way that adding in networking support would be possible without much refactoring.

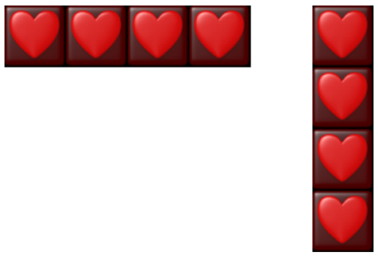
We will be utilizing the GLUT graphics library, Dr. Wani's 2d graphics library, and a driver API for Xbox 360 controllers. The project repository, which includes all source code (and even this document), will be managed on github.com. We will be using the git version control system.

Glossary

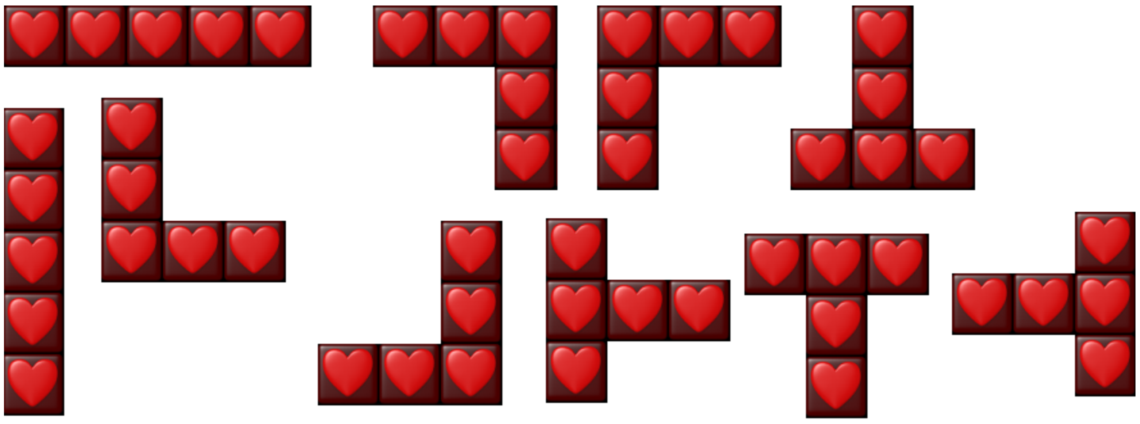
- **Clearing blocks (aka breaking)** – Blocks are cleared whenever a combo occurs. They dissapear, and any blocks above them fall.
- **Clearing garbage blocks -** Breaking garbage blocks *(see Garbage blocks)* is somewhat different. They are indirectly cleared by a combination of normal blocks occuring directly adjacent to them. When a garbage block is cleared, the first (and possibly only) row of the garbage block changes into normal blocks. Any combinations made as these normal blocks fall count towards the chain multiplier *(see Chains)*. If any garbage blocks of the same color are adjacent to a cleared garbage block, they are also cleared in the same fashion.
- **Combination (aka combo)** – Combinations are one of the most basic elements in the game. They occur when 3 or more blocks are in a row, at which point all blocks used to create the combination are cleared. The greater then number of blocks in the combination, the greater the score for clearing them. Note that in some cases, blocks of different types can be used to create a single large combination.
    - **3x Combo**
        - smallest possible combo
        - 3 of a given block type A in one of two possible ways


Every possible 3x combo

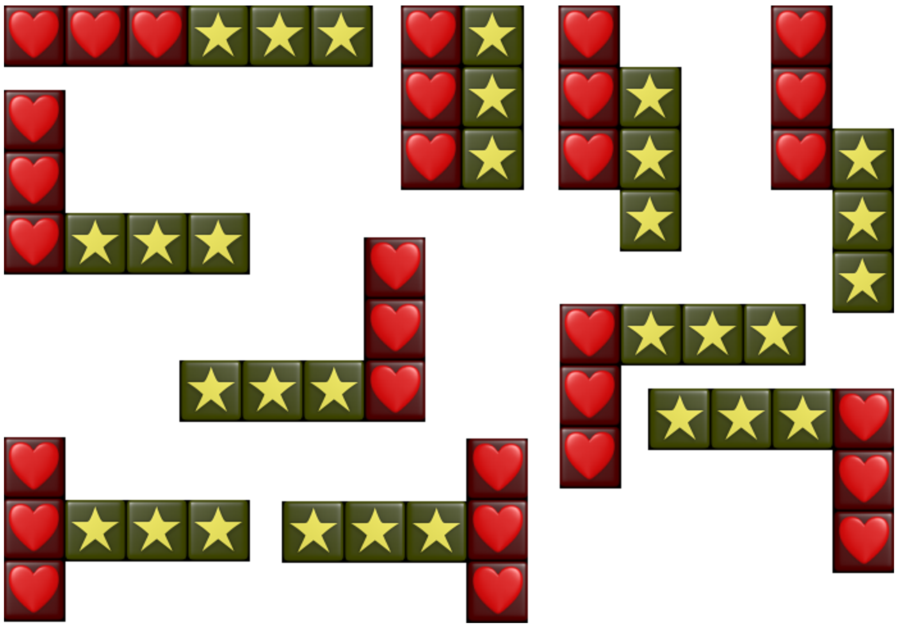
    - **4x Combo**
        - 4 of a given block type A in one of two possible ways


Every possible 4x combo

- **5x Combo**
  - 5 of a given block type A in one of 10 possible ways


Every possible 5x combo

- **6x Combo**
  - 3 of a given block type A and 3 of a given block type B (or A) in one of many possible ways

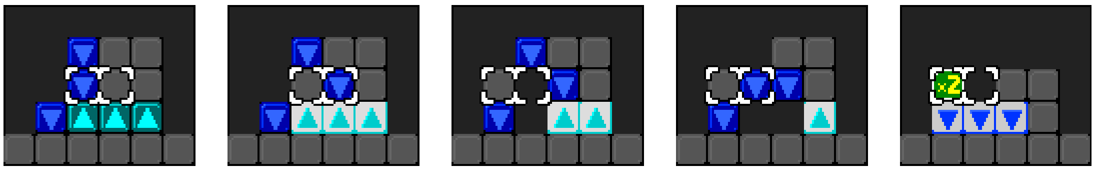

Every possible 6x combo

  -

- **7x Combo**
  - 4 of a block type A and 3 of a block type B (or A) in one of many possible ways
- **8x Combo**
  - 5 of a block type A and 3 of a block type B (or A) or
  - 4 of a block type A and 4 of a block type B (or A)
- **9x Combo**

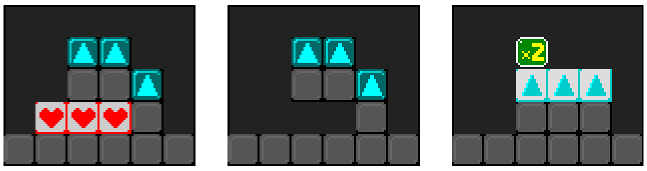
        - 5 of a block type A and 4 of a block type B (or A)
    - **10x Combo**
        - largest possible combo
        - 5 of a block type A and 5 of a block type B (or A)
- **<u>Chains </u>**– Chains of combinations occur when more than one combination occurs as a direct result of blocks falling due to a prior combination. A (theoretically) unlimited length of chains are possible. Chains act as multipliers to the score that would normally be calculated for a combination. Chaining together N amount of combinations result in a score multiple for that run of xN. From the original Tetris Attack, here are the 5 different ways that chains can occur, in order of increasing difficulty:

    - **Basic Chain**



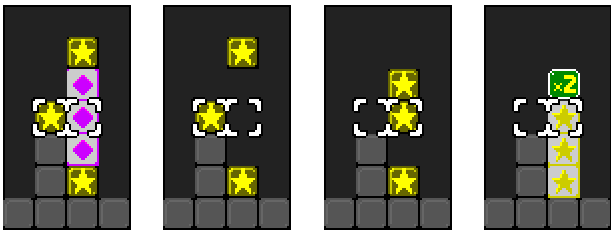
Basic chain

    - **Raised Chain**
        - the first combination is not adjacent to the chained combination

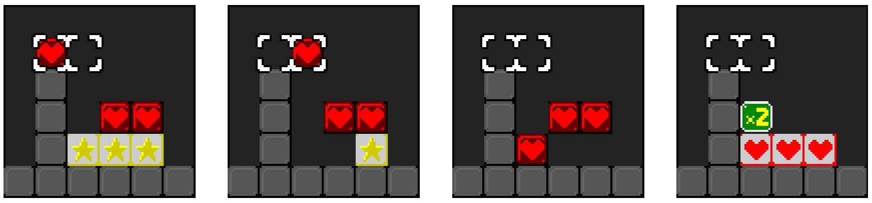


Raised chain

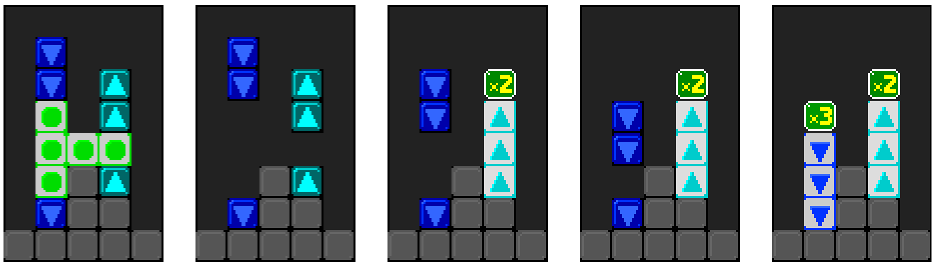
    - **Skill Chain**
        - a chain quickly made by the player by moving a block into the space left from the first combination, creating a new combination once the blocks fall



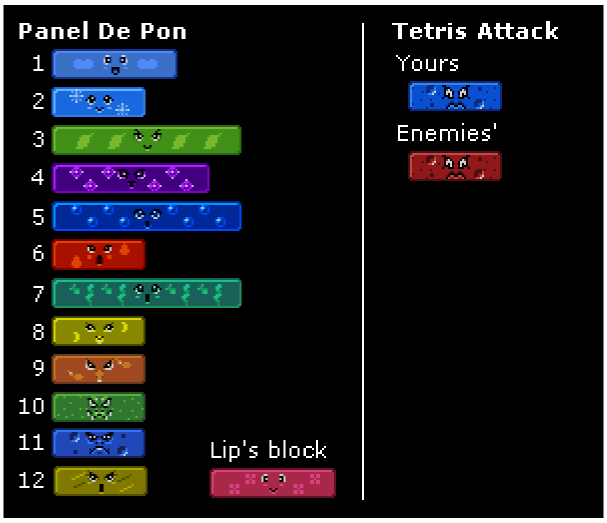
    - *Skill chain*
    - **Advanced Chain**
        - these chains take advantage of the fact that there is a split second before a combination is cleared, where the blocks directly above it have not yet fallen, but it is possible for the player to drop another block


Advanced chain

- **Time-Lag Chain**
  - involve 2 or more combinations being cleared as a direct result of the first combination being cleared


Time-lag chain

- **Garbage blocks** – Special blocks of varying width, which turn into normal blocks when cleared *(see Clearing garbage blocks)*. The sole purpose of garbage blocks is to block players from making combinations, while simultaneously increasing the height of their stack. Garbage blocks fall at random in single player mode. In 2-player mode, garbage blocks fall as a result of chains made by opposing players.

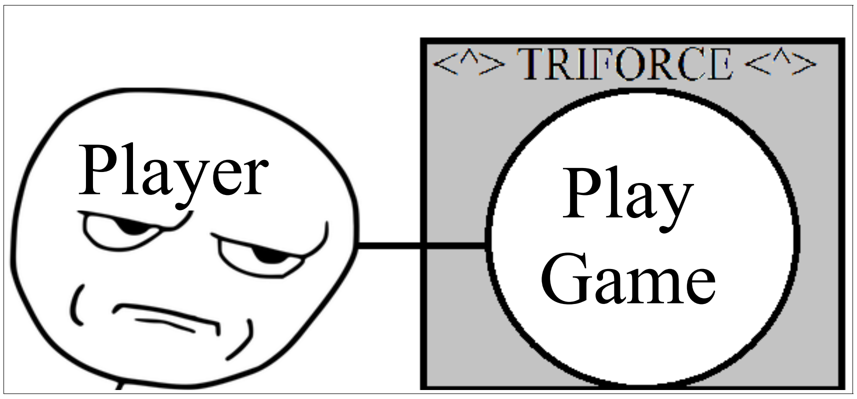


Garbage blocks from Panel De Pon and Tetris Attack

- **Raise blocks** – The blocks naturally rise as time progresses. The player can manually speed up this process if they want more blocks on the screen. It is possible to raise them past the top of the grid, and lose the game.
- **Stack (aka Grid** – The stack consists of the play area, which holds blocks and the cursor.
- **Swapping blocks** – The only operation that can be directly performed on blocks. The swap in the left part of the cursor swaps places with the block in the right part of the cursor. Other than moving the cursor and raising the stack *(see Raising the stack)*, there is no other input

action that can be performed in-game.

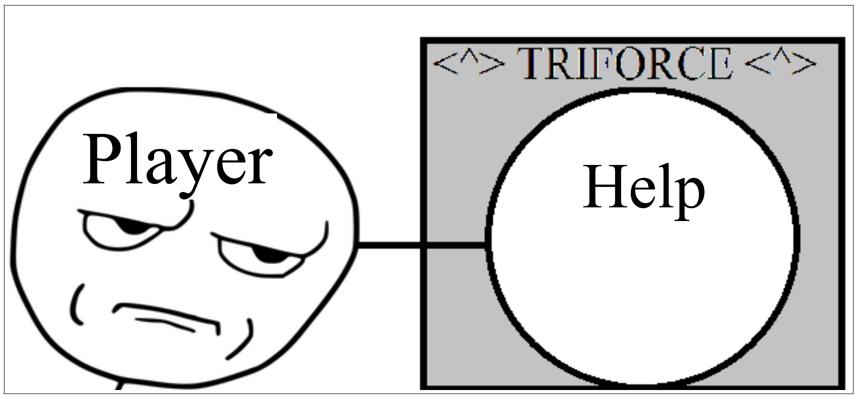
Revised Requirements



Brief Description: When the Player activates the Play Game menu item, the game state changes to Play, and the game starts.

Step-by-Step Description:

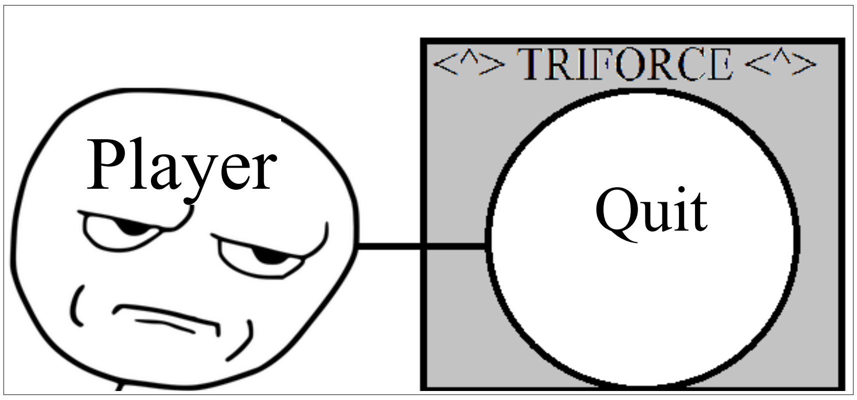
6. Load Grid, Cursor, and blocks
7. Countdown 3... 2... 1...
8. Start game



Brief Description: When the Player activates the Help menu item, the game displays instructions of the various actions that are possible in the game, and graphics of how different combinations/chains are made.

Step-by-Step Description:

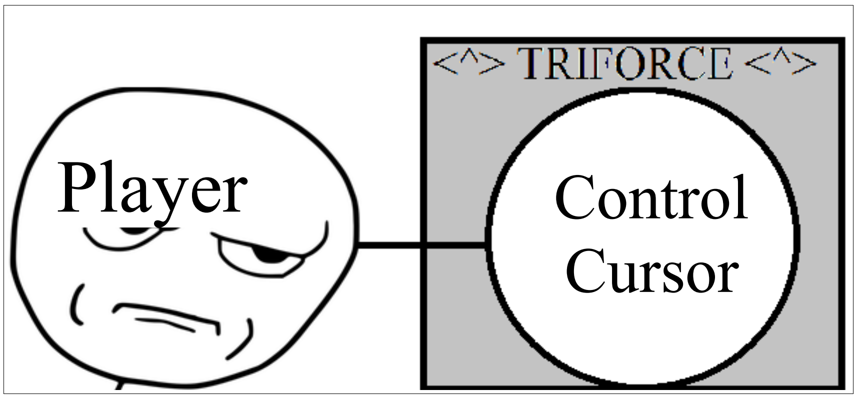
1. Load and display Help information, which is hard-coded text and images
2. Return to menu



Brief Description: When the Player activates the Quit menu item, the game asks the user if they are sure, and then quits upon 'yes', or goes back to the menu upon 'no'.

Step-by-Step Description:

1. Display dialog asking user to confirm that they would like to quit
2. Quit (if user selected yes) or return to menu (if user selected no)

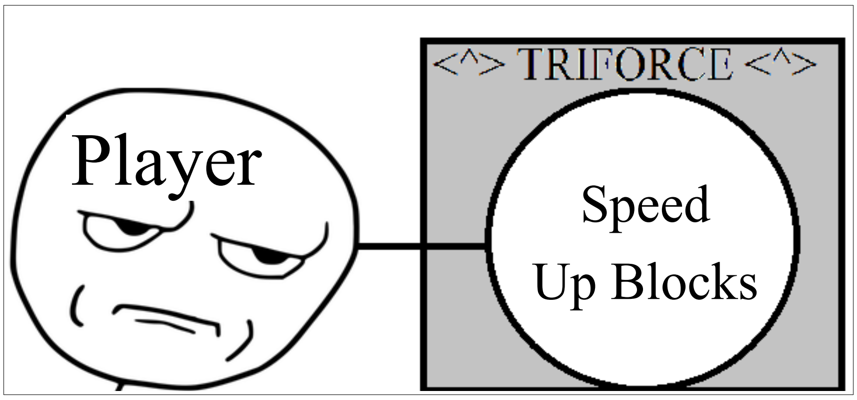


Brief Description: In the Play state, the user has control of a special in-game cursor for selecting two blocks at a time. The cursor can move up, down, left, and right. When the Player activates the Swap Blocks function, the cursor swaps the two blocks that are underneath it.

Step-by-Step Description:

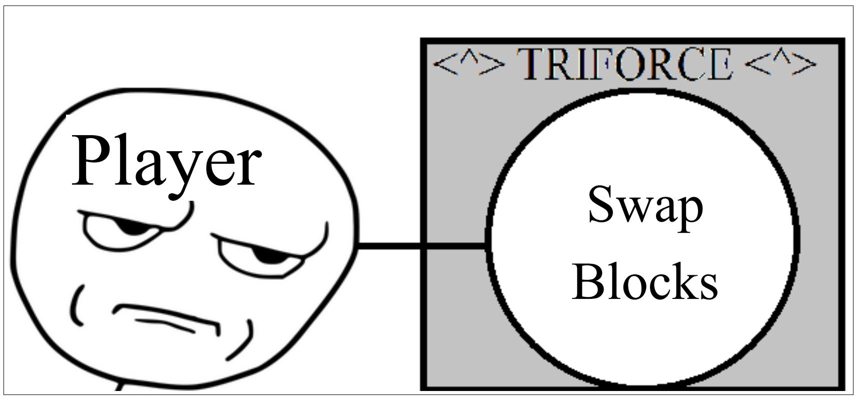
1. For each input action, attempt to move the Cursor accordingly.
2. If the user pressed Up, go up unless the Cursor is at the top of the grid already. If the user pressed Down, go down unless the block below the current block is not enabled yet (i.e. It's still being pushed onto the grid). If the user pressed left or right, move the Cursor left or right, unless the Cursor is already on the edge of the grid. If the user pressed swap, then swap the blocks selected by the cursor.
3. Clear blocks if a combination was made in the last step.
4. Add points to score if the current chain has ended.
5. Go back to step 1.

Brief Description: In the Play state, when the Player activates the Speed Up Blocks function, the Grid will speed up it's normal action of moving the blocks up.



Step-by-Step Description:

1. Move the stack up a larger amount than normal
2. If the Speed Up Blocks button is still being pressed, goto 1, otherwise continue raising stack at the normal rate.



Step-by-Step Description:

1. Exchange the block in the right side of the cursor with the block in the left side.

Section 2: Analysis workflow

2.1 Analysis workflow

2.1.1 Functional Modeling

I. Normal scenario
   1. A player awakens, drenched in sweat. He knows that his addiction to Triforce is more severe than he would ever admit to his psychiatrist, but he doesn't care. After releasing an ominous battle moan, he throws on a robe, shuffles into his slippers, and makes his way towards the computer. As the program loads, the neon glow of the play button flashes across his face, like the reflection of the sun upon the moon.
   2. The player clicks play.
   3. A player presses the Play button, to start a single-player game.
   4. The menu unloads.
   5. The game loads.
   6. A timer counts down from 3 to 1, then says GO.
   7. The game begins.
   8. Blocks begin rising from the bottom. The elapsed game play timer starts.
   9. The player moves the block cursor around with the keyboard, and activates the swap block action.
   10. Blocks swap.
   11. Combo detection is activated, and a combo is detected.
   12. The rising of the blocks stops, and the time at which it stopped is recorded.
   13. Blocks involved in the combo dissapear.
   14. Fall detection is activated, and the blocks above the combo fall.
   15. The blocks hit the bottom.
   16. Combo detection is activated, and another combo is detected.
   17. The chaining of the two combos is detected.
   18. Steps 10-13 repeat.
   19. The chain ends, and the blocks start rising again.
   20. The score is added, with a multiplier based on the number of chains (2).
   21. The player presses the button bound to Speed Up Blocks for too long, and the stack reaches the top.
   22. The player loses.
   23. The score is displayed, and the user clicks OK
   24. The main menu loads and the game objects are deallocated.
   25. The user clicks quit.
   26. The game exits.

II. Normal scenario – informational
   1. A player presses the Help button.
   2. Images and text explaining how to play the game are loaded.
   3. After reviewing the information, the user clicks quit
   4. The game exits.

III. Exception scenario
   1. A player presses the Play button, to start a single-player game.
   2. The menu unloads.
   3. The game loads.
   4. A timer counts down from 3 to 1, then says GO.

5. The game begins.
6. Blocks begin rising from the bottom. The elapsed game play timer starts.
7. The user has to leave the computer, and clicks the Pause button.
8. A menu loads.
9. The user returns to the computer, but decides not to resume the game and clicks Quit.
10. The game exits.

2.1.2 Entity Class Modeling

The **Triforce** **main menu** loads and several **buttons** are displayed. The **player** selects an **button** which performs an **action**, using an **input device** such as the **keyboard**, **mouse**, or an **xbox controller**. The **user** also has option of selecting help from the **menu**, which loads a help screen. The user can click the quit **button**, and the **game** will exit. If the user selects the play button, the **gameplay** begins. Then a **grid** displays, filled with **blocks**, and a **HUD** is displayed. The user moves the special in-game **cursor** around, and swaps blocks. The user creates **combinations** with the blocks, and a **fall event** occurs. More than one combination after another creates **chains**.

Nouns

Triforce
main menu
buttons player
button
action
input device
keyboard
mouse
xbox controller
user
button
game
gameplay
grid
blocks
cursor
combinations
chains
fall event
HUD

Cursor
pos : Point
row : int
col : int

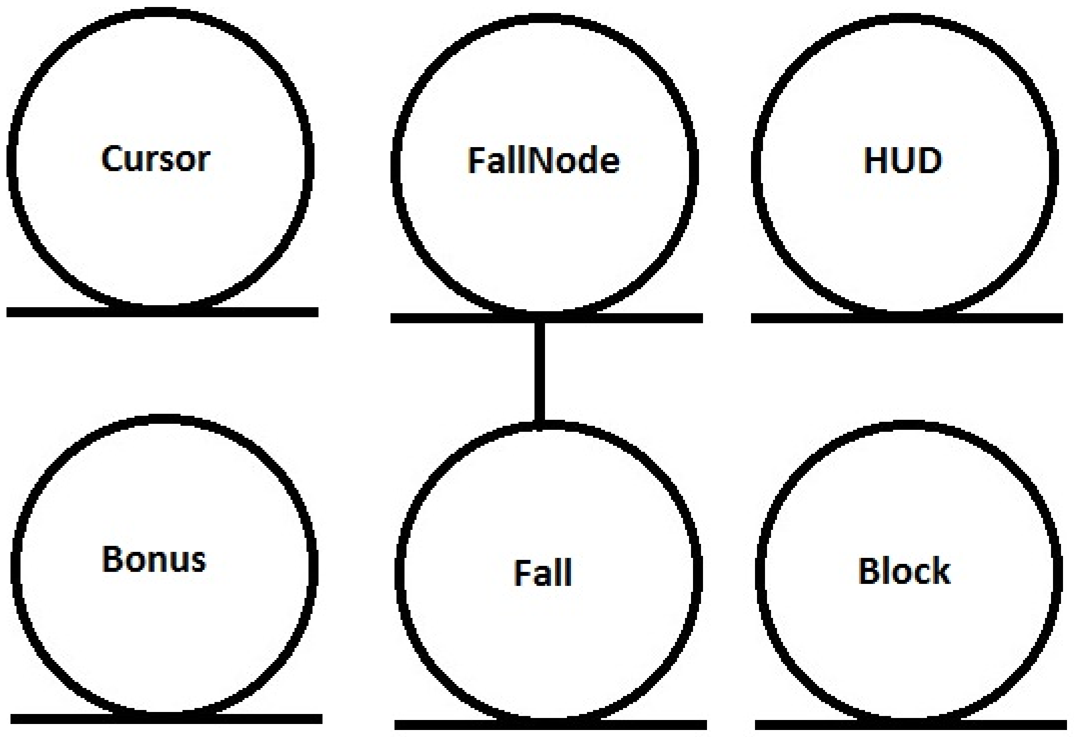
Bonus
col : int
row : int
bonusType : BonusType

FallNode
chainCount : int
enabled : bool
numFalls : int

HUD
currentSeconds : int
score : int
x : float
y: float

Fall
chainCount : int
fallInterval : int

Block
fallFactor : int
fallOffset : int



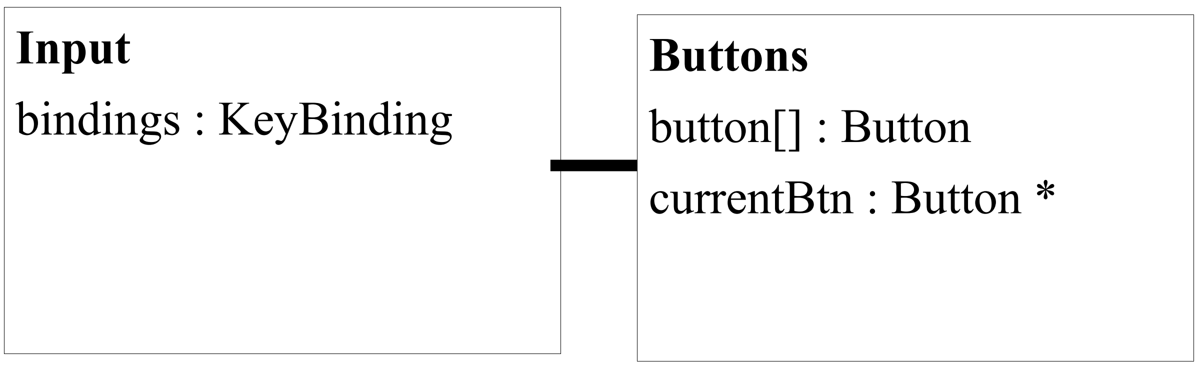
2.1.3 Boundary Class Modeling

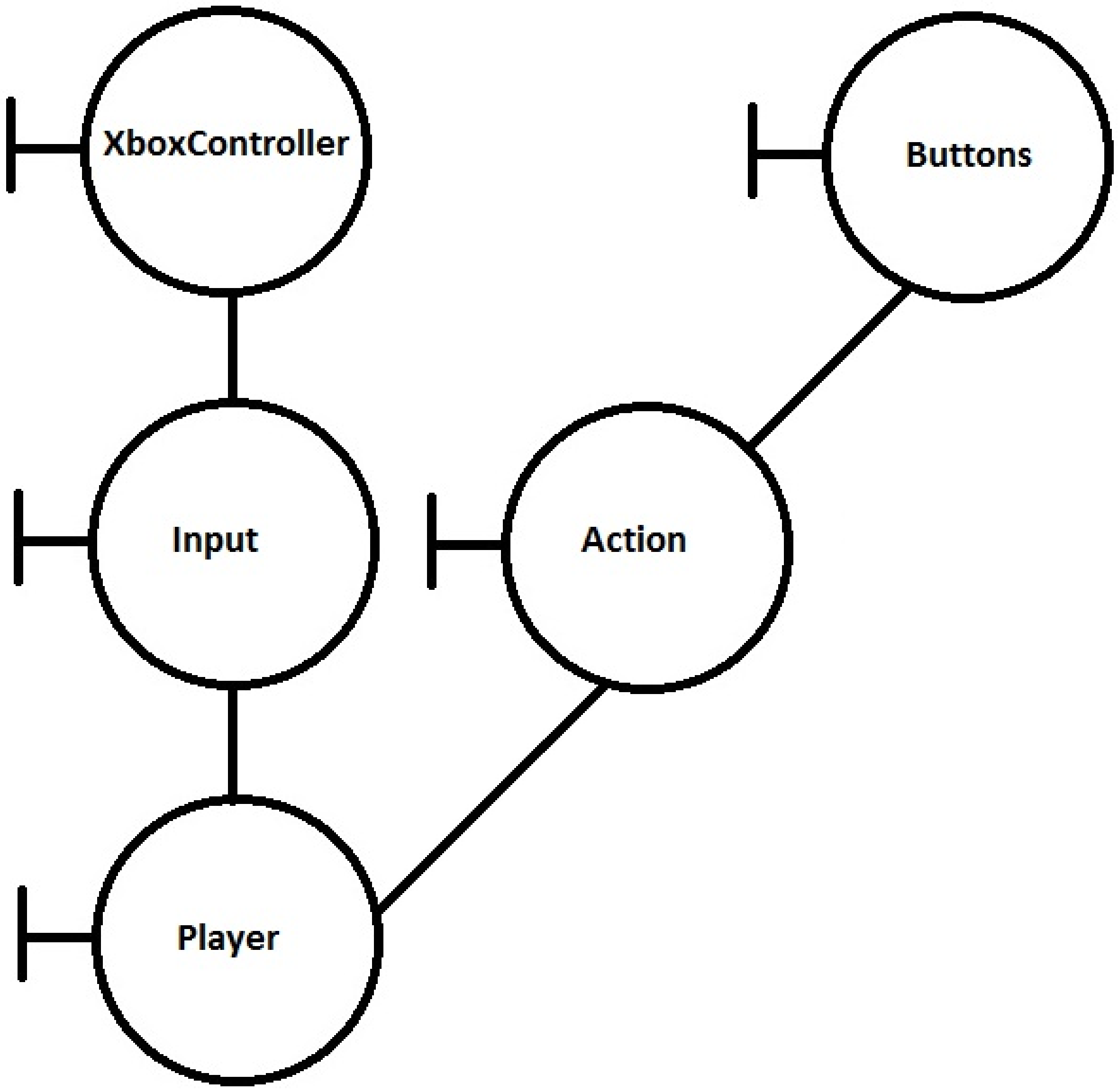
Boundary classes:

Input

Buttons

Action





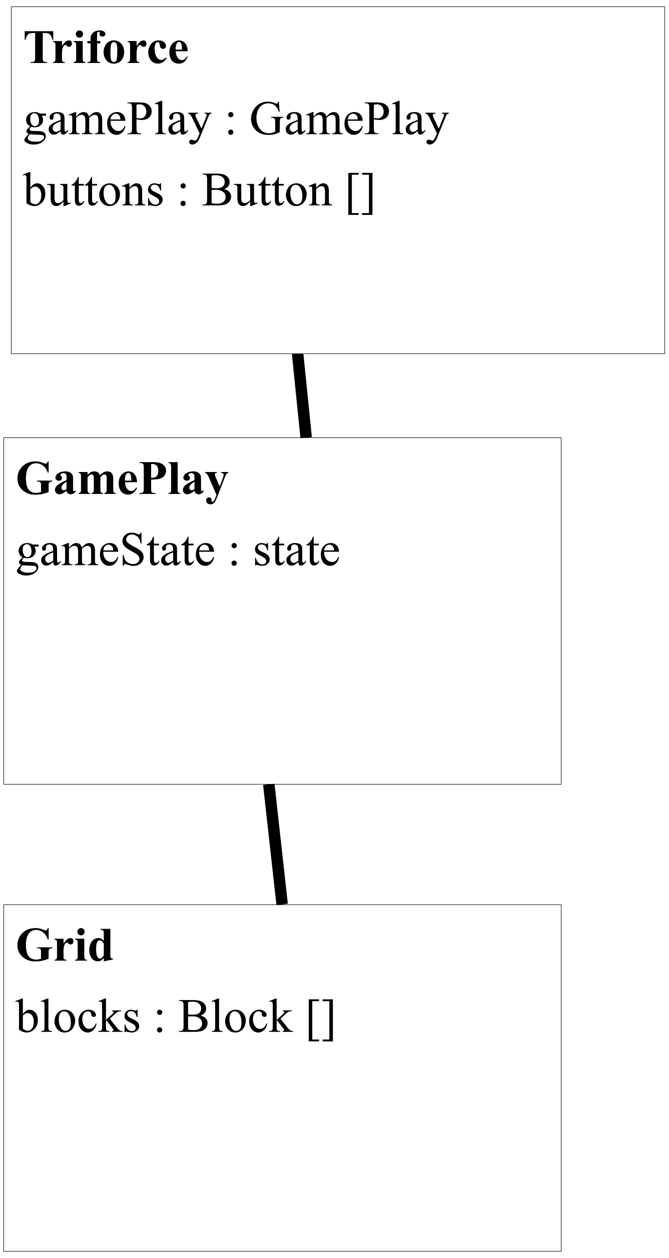
2.1.4 Control Class Modeling

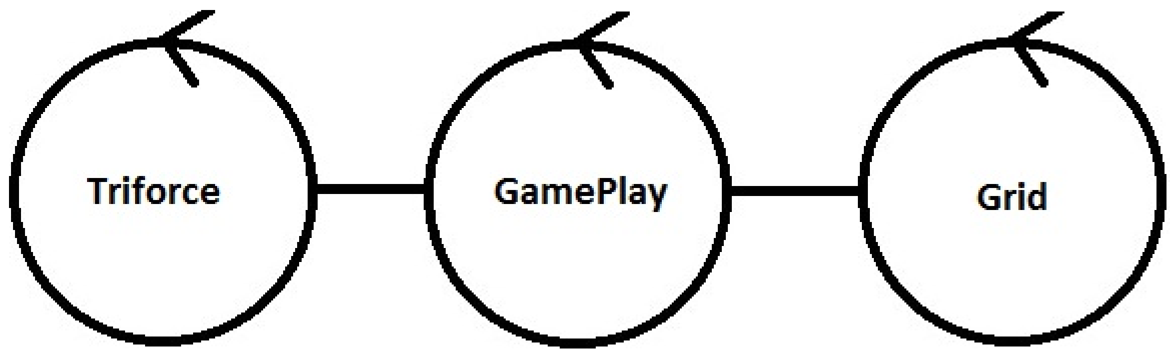
Control classes:

Triforce

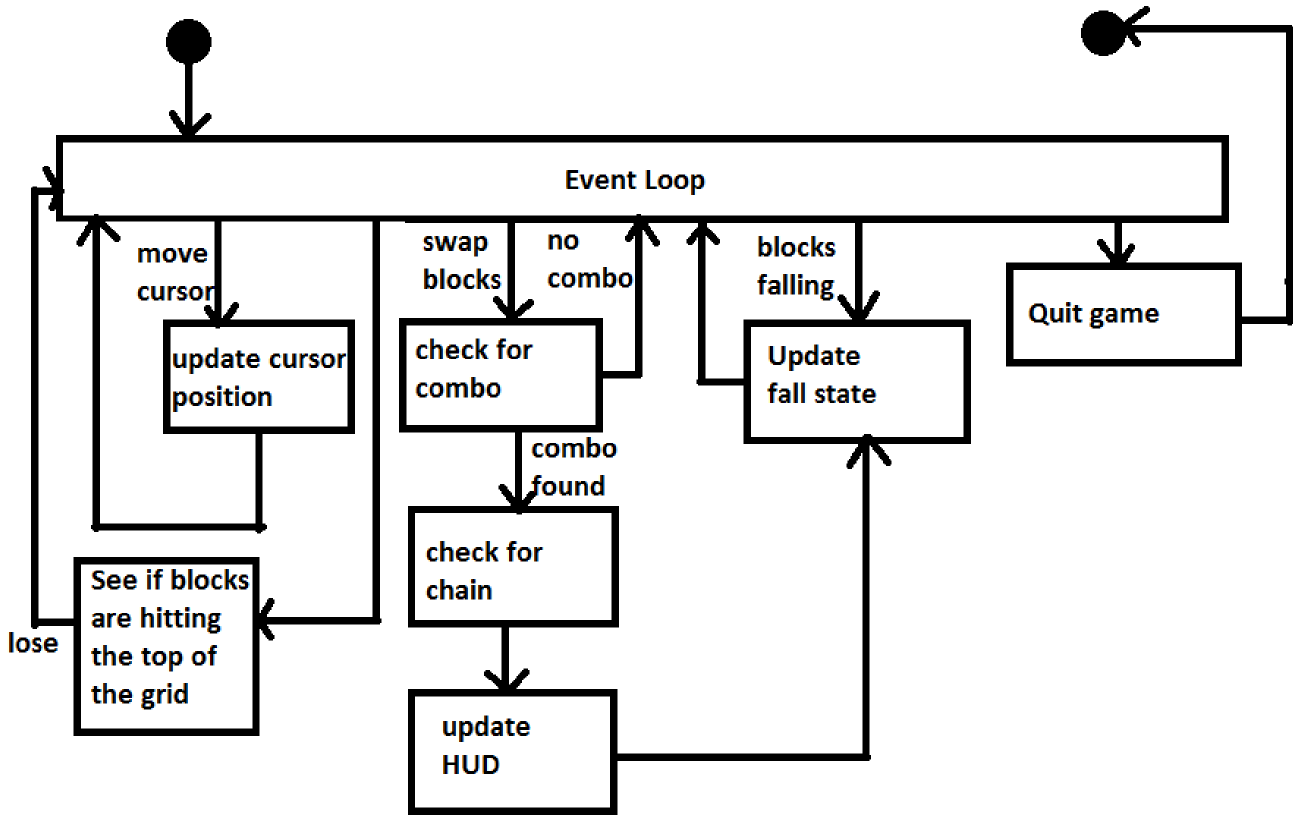
GamePlay

Grid

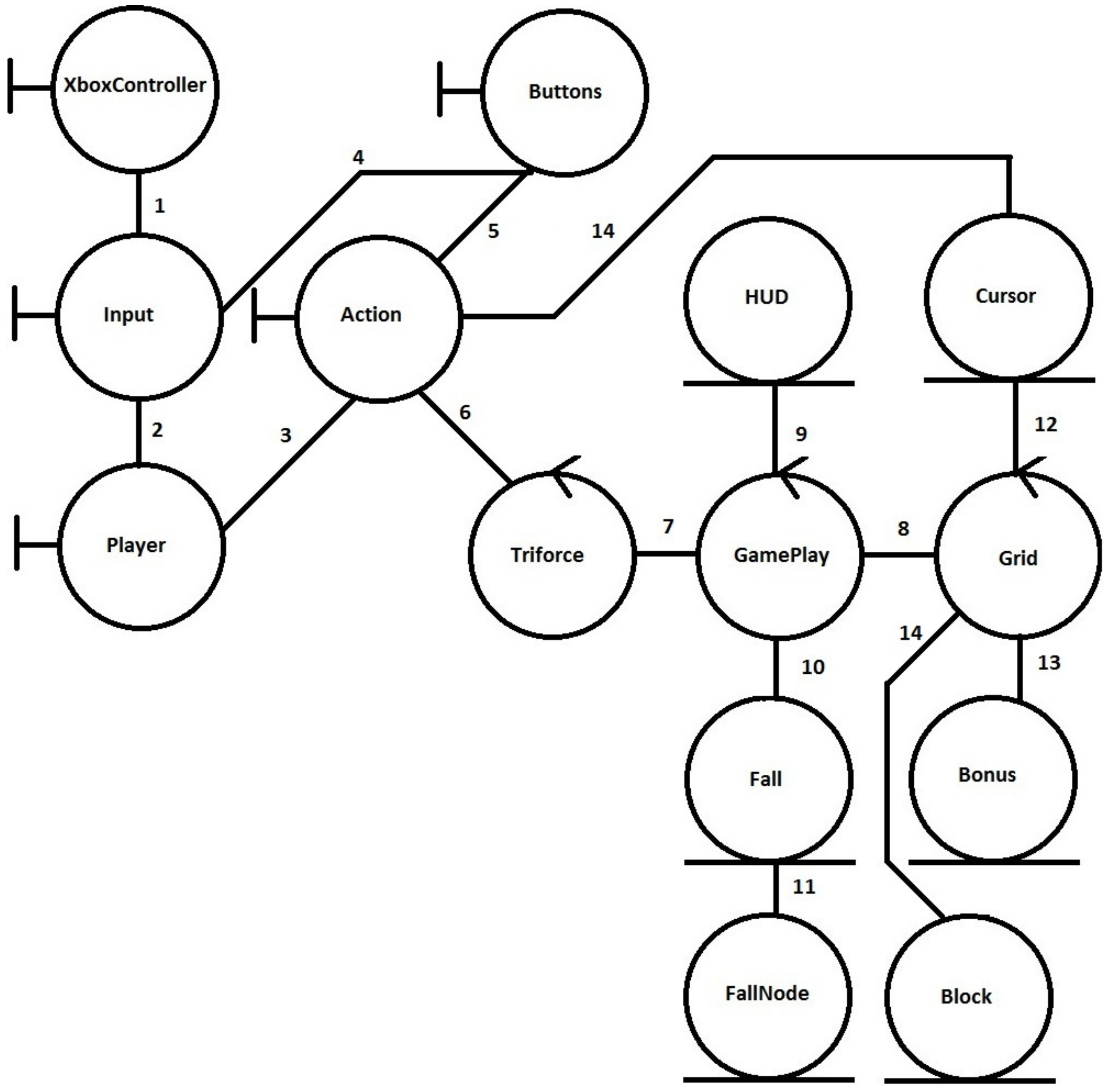



Triforce
GamePlay
Grid

2.1.5 Dynamic Modeling



2.1.6 Communication Modeling



1. Input polls state of xbox controller periodically
2. Sends signals to a player that has input bindings that correspond to actual input
3. Tells action to activate itself
4. Input triggers button event
5. Button even triggers the action it is bound to
6. Action triggers whatever menu event it was registered to
7. Loads actual game
8. Manipulates grid to account for falls

9. Updates score and time
10. Look for fall events and handle them
11. Update fall nodes
12. Cursor manipulates grid speed and tells grid to swap blocks
13. Update bonuses
14. Update block positions

2.3 Walkthrough

Not all data members are shown.

Some things seem to be incomplete.

- Walkthrough by Gabe Pike

2.3 Revised Analysis workflow

2.3.1 Functional Modeling

IV. Normal scenario
   1. A player awakens, drenched in sweat. He knows that his addiction to Triforce is more severe than he would ever admit to his psychiatrist, but he doesn't care. After releasing an ominous battle moan, he throws on a robe, shuffles into his slippers, and makes his way towards the computer. As the program loads, the neon glow of the play button flashes across his face, like the reflection of the sun upon the moon.
   2. The player clicks play.
   3. A player presses the Play button, to start a single-player game.
   4. The menu unloads.
   5. The game loads.
   6. A timer counts down from 3 to 1, then says GO.
   7. The game begins.
   8. Blocks begin rising from the bottom. The elapsed game play timer starts.
   9. The player moves the block cursor around with the keyboard, and activates the swap block action.
   10. Blocks swap.
   11. Combo detection is activated, and a combo is detected.
   12. The rising of the blocks stops, and the time at which it stopped is recorded.
   13. Blocks involved in the combo dissapear.
   14. Fall detection is activated, and the blocks above the combo fall.
   15. The blocks hit the bottom.
   16. Combo detection is activated, and another combo is detected.
   17. The chaining of the two combos is detected.
   18. Steps 10-13 repeat.
   19. The chain ends, and the blocks start rising again.
   20. The score is added, with a multiplier based on the number of chains (2).
   21. The player presses the button bound to Speed Up Blocks for too long, and the stack reaches the top.
   22. The player loses.
   23. The score is displayed, and the user clicks OK
   24. The main menu loads and the game objects are deallocated.
   25. The user clicks quit.
   26. The game exits.

V. Normal scenario – informational
   1. A player presses the Help button.
   2. Images and text explaining how to play the game are loaded.
   3. After reviewing the information, the user clicks quit
   4. The game exits.

VI. Exception scenario
   1. A player presses the Play button, to start a single-player game.
   2. The menu unloads.
   3. The game loads.
   4. A timer counts down from 3 to 1, then says GO.
   5. The game begins.
   6. Blocks begin rising from the bottom. The elapsed game play timer starts.
   7. The user has to leave the computer, and clicks the Pause button.

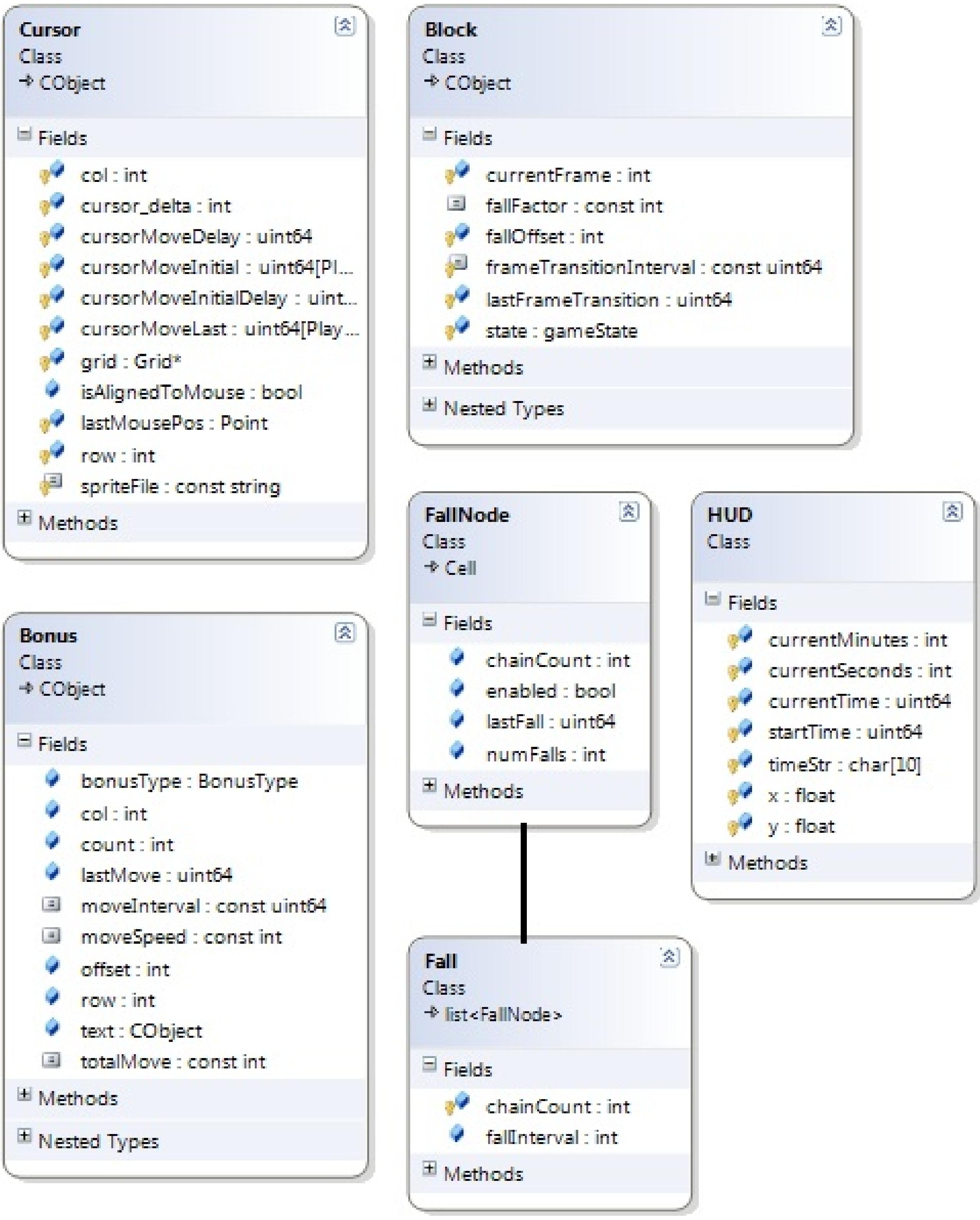
8. A menu loads.
9. The user returns to the computer, but decides not to resume the game and clicks Quit.
10. The game exits.

2.3.2 Entity Class Modeling

The **Triforce** **main menu** loads and several **buttons** are displayed. The **player** selects an **button** which performs an **action**, using an **input device** such as the **keyboard**, **mouse**, or an **xbox controller**. The **user** also has option of selecting help from the **menu**, which loads a help screen. The user can click the quit **button**, and the **game** will exit. If the user selects the play button, the **gameplay** begins. Then a **grid** displays, filled with **blocks**, and a **HUD** is displayed. The user moves the special in-game **cursor** around, and swaps blocks. The user creates **combinations** with the blocks, and a **fall event** occurs. More than one combination after another creates **chains**.

Nouns

Triforce
main menu
buttons player
button
action
input device
keyboard
mouse
xbox controller
user
button
game
gameplay
grid
blocks
cursor
combinations
chains
fall event
HUD



Cursor
Class
→ CObject

Fields
- col : int
- cursor_delta : int
- cursorMoveDelay : uint64
- cursorMoveInitial : uint64[Pl...
- cursorMoveInitialDelay : uint...
- cursorMoveLast : uint64[Play...
- grid : Grid*
- isAlignedToMouse : bool
- lastMousePos : Point
- row : int
- spriteFile : const string

Methods

Block
Class
→ CObject

Fields
- currentFrame : int
- fallFactor : const int
- fallOffset : int
- frameTransitionInterval : const uint64
- lastFrameTransition : uint64
- state : gameState

Methods

Nested Types

FallNode
Class
→ Cell

Fields
- chainCount : int
- enabled : bool
- lastFall : uint64
- numFalls : int

Methods

HUD
Class

Fields
- currentMinutes : int
- currentSeconds : int
- currentTime : uint64
- startTime : uint64
- timeStr : char[10]
- x : float
- y : float

Methods

Bonus
Class
→ CObject

Fields
- bonusType : BonusType
- col : int
- count : int
- lastMove : uint64
- moveInterval : const uint64
- moveSpeed : const int
- offset : int
- row : int
- text : CObject
- totalMove : const int

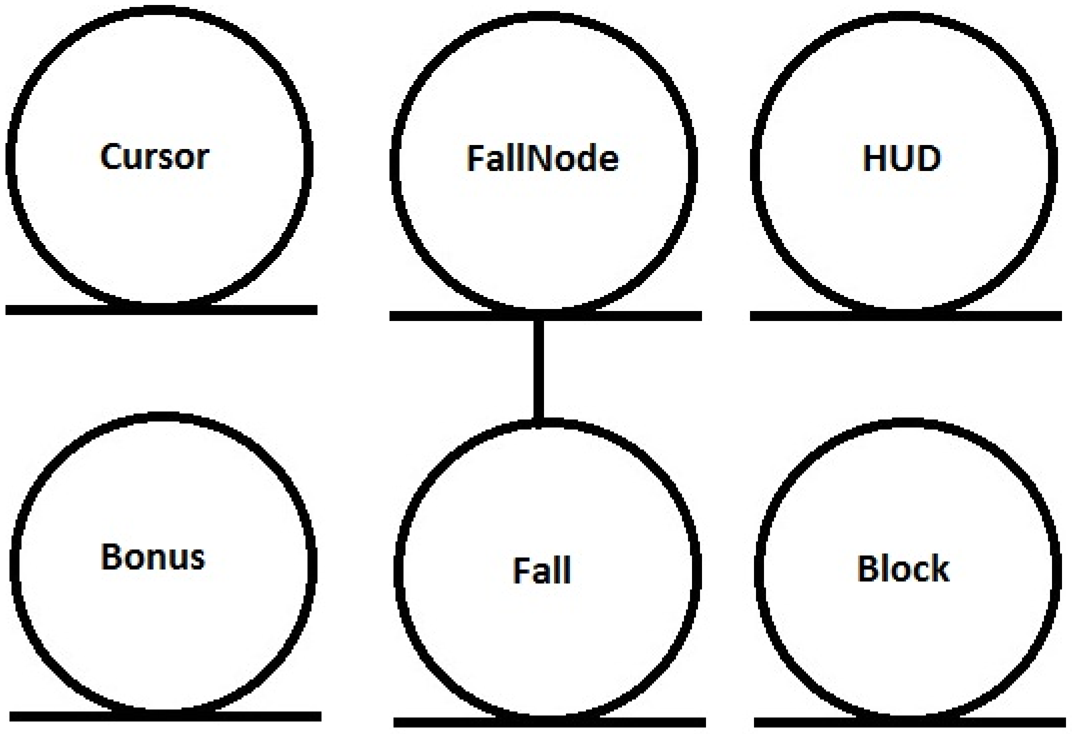
Methods

Nested Types

Fall
Class
→ list<FallNode>

Fields
- chainCount : int
- fallInterval : int

Methods


Cursor
FallNode
HUD
Bonus
Fall
Block

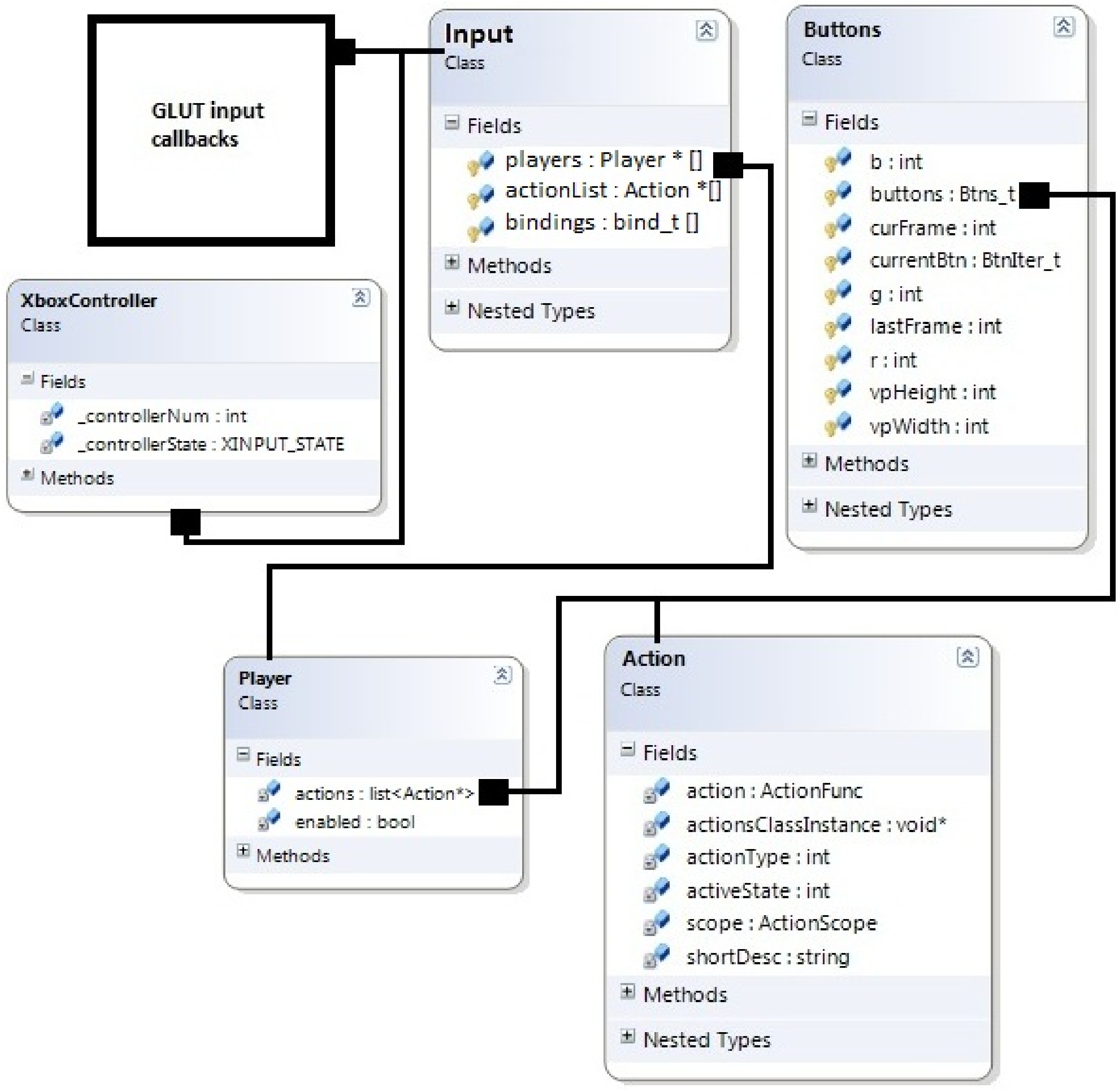
2.3.3 Boundary Class Modeling

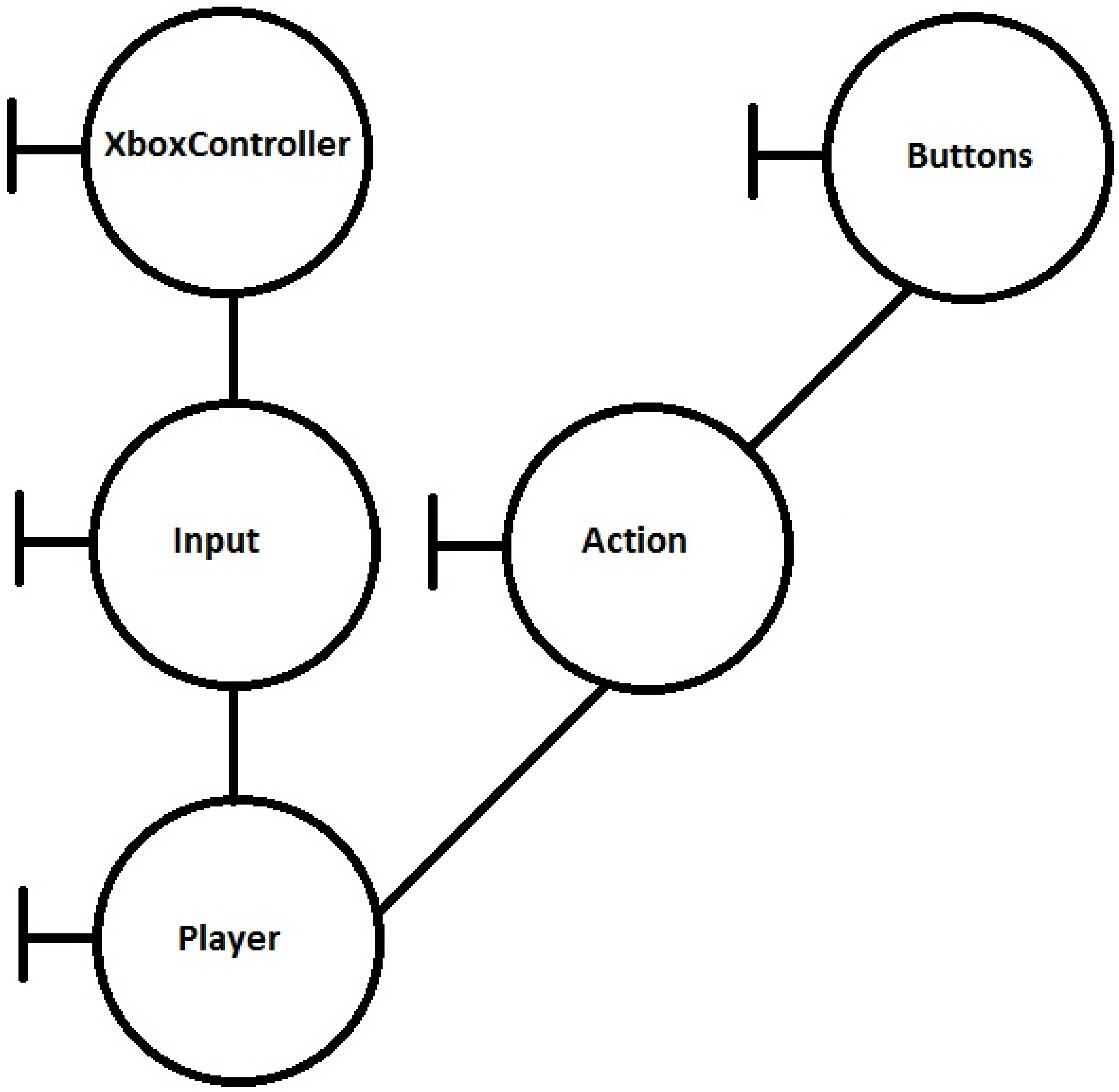
Boundary classes:

Input

Buttons

Action




XboxController
Buttons
Input
Action
Player

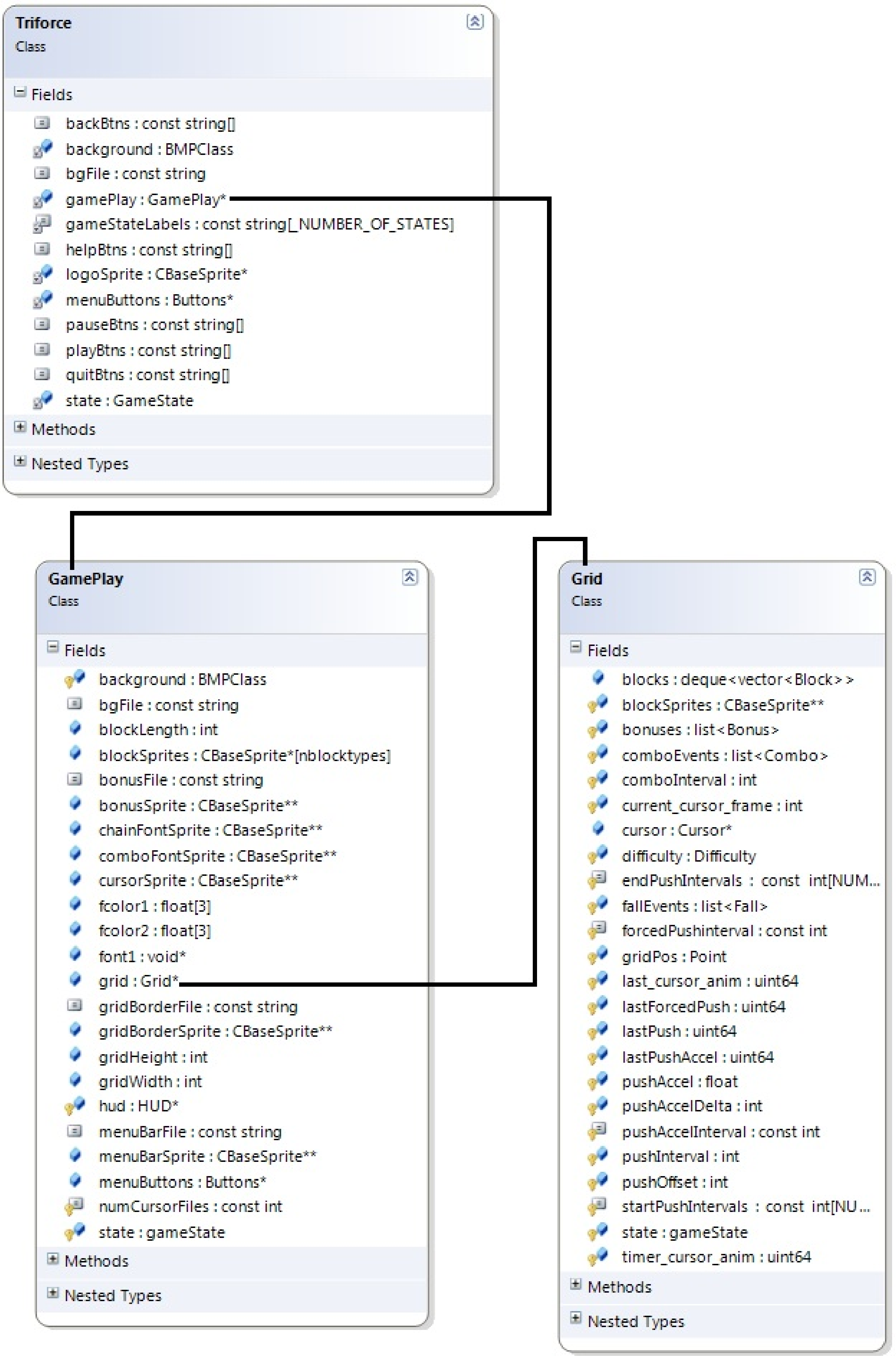
2.3.4 Control Class Modeling

Control classes:

Triforce

GamePlay

Grid



Triforce
Class

Fields
- backBtns : const string[]
- background : BMPClass
- bgFile : const string
- gamePlay : GamePlay*
- gameStateLabels : const string[_NUMBER_OF_STATES]
- helpBtns : const string[]
- logoSprite : CBaseSprite*
- menuButtons : Buttons*
- pauseBtns : const string[]
- playBtns : const string[]
- quitBtns : const string[]
- state : GameState

Methods

Nested Types

GamePlay
Class

Fields
- background : BMPClass
- bgFile : const string
- blockLength : int
- blockSprites : CBaseSprite*[nblocktypes]
- bonusFile : const string
- bonusSprite : CBaseSprite**
- chainFontSprite : CBaseSprite**
- comboFontSprite : CBaseSprite**
- cursorSprite : CBaseSprite**
- fcolor1 : float[3]
- fcolor2 : float[3]
- font1 : void*
- grid : Grid*
- gridBorderFile : const string
- gridBorderSprite : CBaseSprite**
- gridHeight : int
- gridWidth : int
- hud : HUD*
- menuBarFile : const string
- menuBarSprite : CBaseSprite**
- menuButtons : Buttons*
- numCursorFiles : const int
- state : gameState

Methods

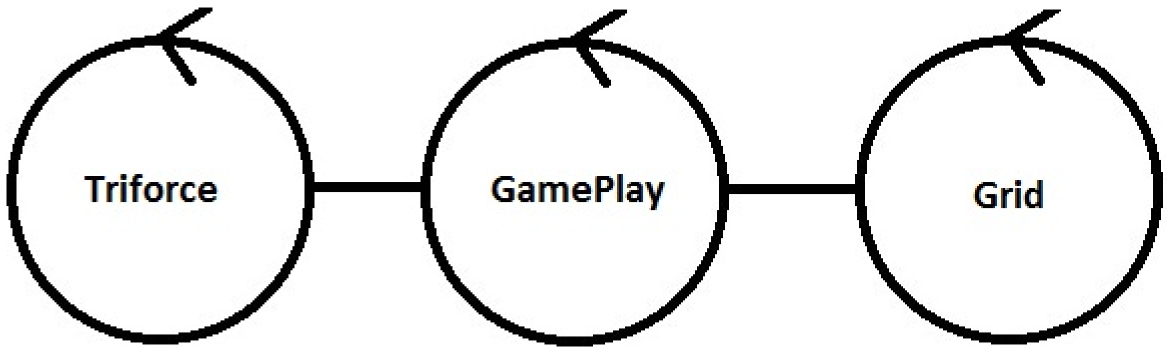
Nested Types

Grid
Class

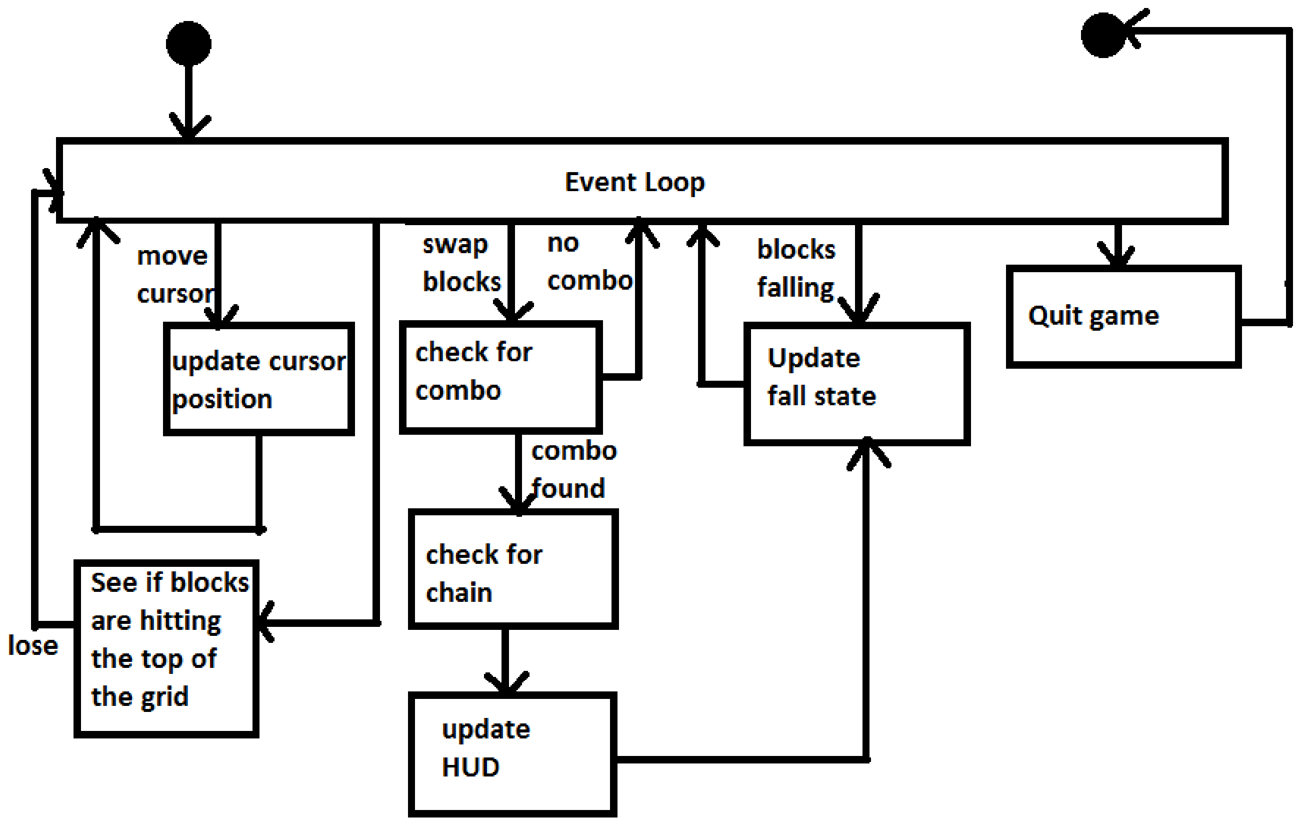
Fields
- blocks : deque<vector<Block>>
- blockSprites : CBaseSprite**
- bonuses : list<Bonus>
- comboEvents : list<Combo>
- comboInterval : int
- current_cursor_frame : int
- cursor : Cursor*
- difficulty : Difficulty
- endPushIntervals : const int[NUM...
- fallEvents : list<Fall>
- forcedPushinterval : const int
- gridPos : Point
- last_cursor_anim : uint64
- lastForcedPush : uint64
- lastPush : uint64
- lastPushAccel : uint64
- pushAccel : float
- pushAccelDelta : int
- pushAccelInterval : const int
- pushInterval : int
- pushOffset : int
- startPushIntervals : const int[NU...
- state : gameState
- timer_cursor_anim : uint64

Methods

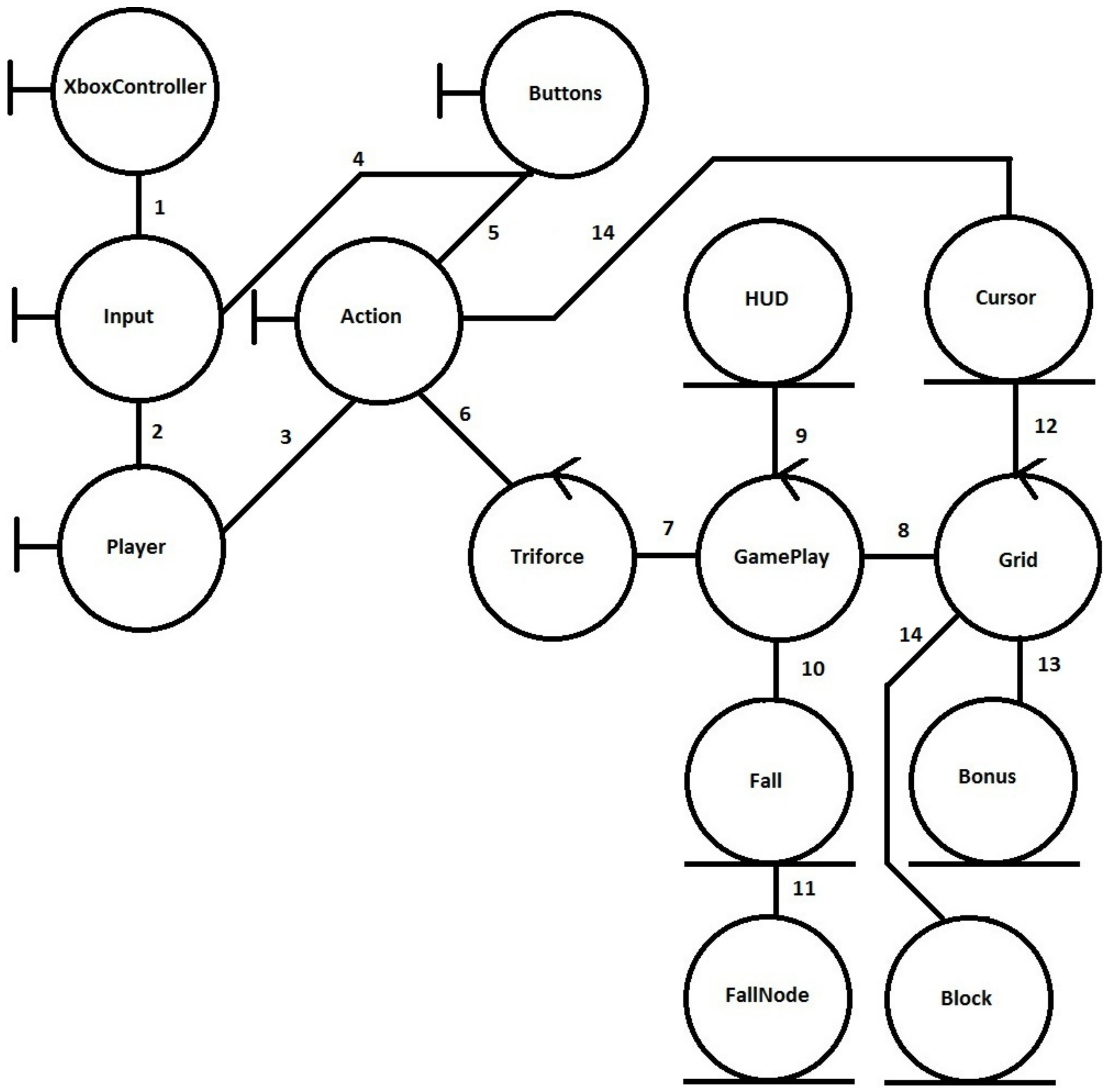
Nested Types


Triforce
GamePlay
Grid

2.3.5 Dynamic Modeling



2.3.6 Communication Modeling



1. Input polls state of xbox controller periodically
2. Sends signals to a player that has input bindings that correspond to actual input
3. Tells action to activate itself
4. Input triggers button event
5. Button even triggers the action it is bound to
6. Action triggers whatever menu event it was registered to
7. Loads actual game
8. Manipulates grid to account for falls
9. Updates score and time

10. Look for fall events and handle them
11. Update fall nodes
12. Cursor manipulates grid speed and tells grid to swap blocks
13. Update bonuses
14. Update block positions

Section 3: Design Workflow

3.1 Design Workflow

3.1.1 Component design

MODULE

Main Loop

RESPONSIBILITY

1. Call main display function
2. Call Input module's input queue handling routine

COLLABORATION

1. Input
2. Triforce

CLASS

Triforce (loader)

RESPONSIBILITY

3. Display startup screen
4. Control state of game

COLLABORATION

3. Buttons
4. Input
5. GamePlay

MODULE

Input

RESPONSIBILITY

1. Accept input from multiple devices
2. Register input actions for different parts of game
3. Bind actions to specific input buttons
4. Queue up input commands and call actions.

COLLABORATION

1. Action (subclass)
2. Player (subclass)

MODULE

Action

RESPONSIBILITY

1. Compare actions
2. Call action callback

COLLABORATION

1. Input module

MODULE

Player

RESPONSIBILITY

1. Store control bindings for each player

COLLABORATION

1. Input module

MODULE

GamePlay

RESPONSIBILITY

1. Load grid
2. Pause game
3. Quit game

COLLABORATION

1. Grid

CLASS

Grid

RESPONSIBILITY

1. Swap blocks
2. Detect chains and combos
3. Move blocks up

COLLABORATION

1. Blocks

CLASS

Cursor

RESPONSIBILITY

1. Move special cursor around
2. Update mouse cursor

COLLABORATION

1. Input
2. Grid

CLASS

Block

RESPONSIBILITY

1. Detect adjacent blocks
2. Set state of block

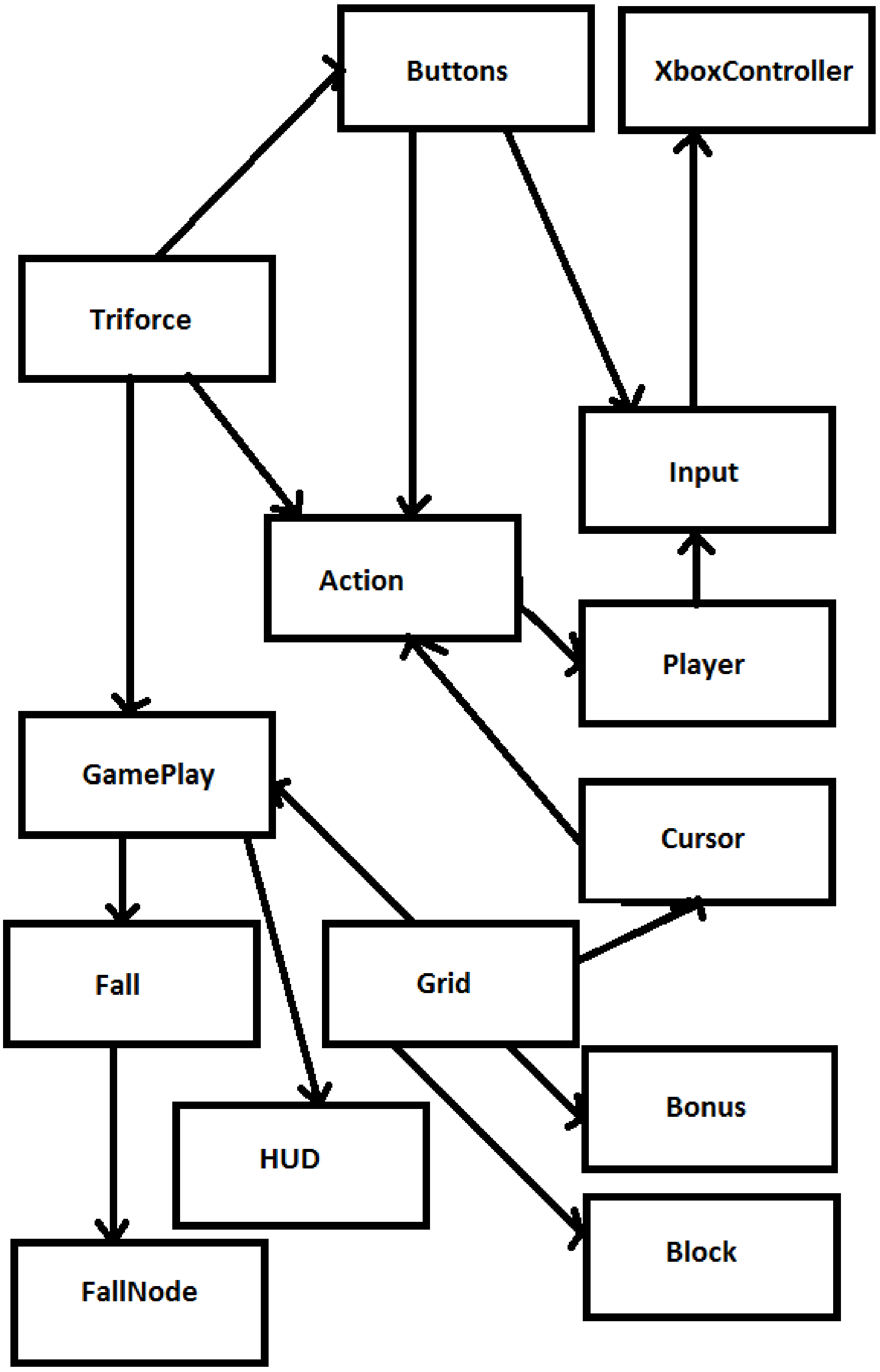
COLLABORATION

2. Grid

3.1.2 Interface design

Actions	**Objects**	**Events**
display()	Handles display of entire game by calling the appropriate display method.	When the users changes the state of the game, the display created by the display routines is changed as an effect.
Input::handleInput()	Handles all input for the entire game.	Any input that the user makes is queued in the input system. Queued actions are actually performed when the display loop calls the handleInput routine.
changeState()	Changes the current state of the game.	Change state events occur when the user presses buttons or loses the game.
PushRow()	Pushes a new row of blocks up from below.	Called periodically at a slow rate, or more quickly if the user presses the button to speed it up.
SwapBlocks()	Changes positions of the two blocks on the grid that are currently selected by the special cursor.	Called only when the user presses a button to swap blocks.

3.1.3 Architectural design



3.1.4 Detailed Class Design

3.2 Walkthrough

Your detailed class diagram is missing

You should include fall-related classes.

--Gabe Pike

3.3 Revised Design After Walkthrough

3.3.1 Component design

MODULE

Main Loop

RESPONSIBILITY

5. Call main display function
6. Call Input module's input queue handling routine

COLLABORATION

6. Input
7. Triforce

CLASS

Triforce (loader)

RESPONSIBILITY

7. Display startup screen
8. Control state of game

COLLABORATION

8. Buttons
9. Input
10. GamePlay

MODULE

Input

RESPONSIBILITY

5. Accept input from multiple devices
6. Register input actions for different parts of game
7. Bind actions to specific input buttons
8. Queue up input commands and call actions.

COLLABORATION

3. Action (subclass)
4. Player (subclass)

MODULE

Action

RESPONSIBILITY

3. Compare actions
4. Call action callback

COLLABORATION

2. Input module

MODULE

Player

RESPONSIBILITY

2. Store control bindings for each player

COLLABORATION

2. Input module

MODULE

GamePlay

RESPONSIBILITY

4. Load grid
5. Pause game
6. Quit game

COLLABORATION

2. Grid

CLASS

Grid

RESPONSIBILITY

4. Swap blocks
5. Detect chains and combos
6. Move blocks up

COLLABORATION

3. Blocks

CLASS

Block

RESPONSIBILITY

3. Detect adjacent blocks
4. Set state of block

COLLABORATION

4. Grid

CLASS

Cursor

RESPONSIBILITY

1. Move special cursor around
2. Update mouse cursor

COLLABORATION

1. Input
2. Grid

CLASS

Fall

RESPONSIBILITY

1. Handle fall events
2. Manage chains

COLLABORATION

1. Grid
2. Block
3. Fallnode

CLASS

FallNode

RESPONSIBILITY

1. Handle a single fall column

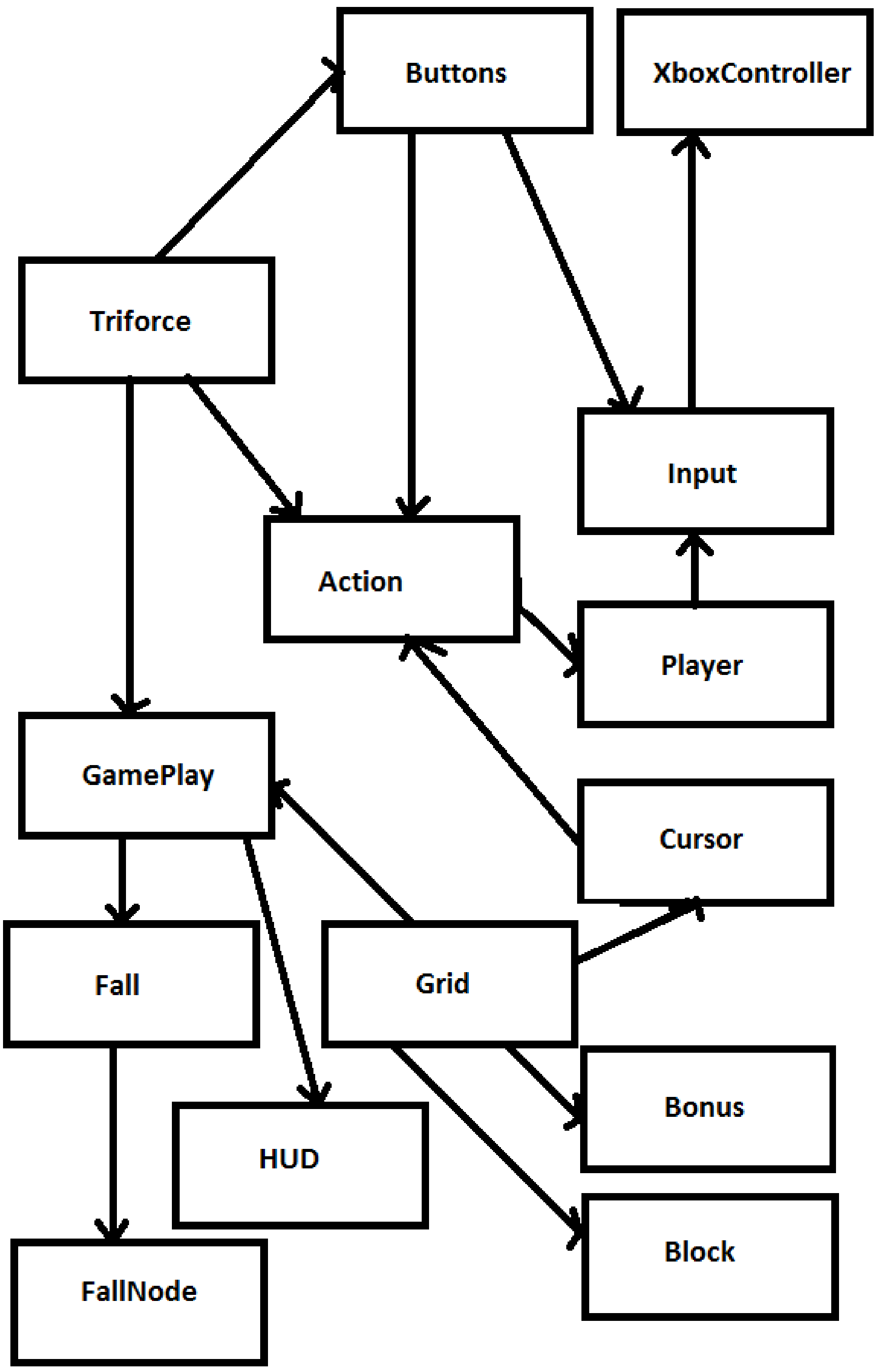
COLLABORATION

1. Fall
2. Grid
3. Block

3.3.2 Interface design

Actions	Objects	Events
display()	Handles display of entire game by calling the appropriate display method.	When the users changes the state of the game, the display created by the display routines is changed as an effect.
Input::handleInput()	Handles all input for the entire game.	Any input that the user makes is queued in the input system. Queued actions are actually performed when the display loop calls the handleInput routine.
changeState()	Changes the current state of the game.	Change state events occur when the user presses buttons or loses the game.
PushRow()	Pushes a new row of blocks up from below.	Called periodically at a slow rate, or more quickly if the user presses the button to speed it up.
SwapBlocks()	Changes positions of the two blocks on the grid that are currently selected by the special cursor.	Called only when the user presses a button to swap blocks.

3.3.3 Architectural design



3.3.4 Detailed Class Design

```
class XboxController {
private:
        XINPUT_STATE _controllerState;
        int _controllerNum;
public:

        virtual ~XboxController (void){
                if(isConnected())
                        Vibrate(0,0);
        }

        XboxController(int player = 0);
        XINPUT_STATE GetState();
        bool isConnected();
        void Vibrate(int left=0, int right=0);
};

class Buttons
{
protected:
        class Button : public CObject
        {
        public:
                bool hovering;
                bool pressing;
                bool enabled;

                Button(CBaseSprite *sprite, int xpos, int ypos);
                void unhover();
                void hover();
                void activate();
                void display();
                int getFrameNum() const;


                void * actionClassInstance;
                int actionArg;
                void (*action)(void *classInstance, int actionArg);
                void disable() {enabled = hovering = pressing = false;}
                void enable() {enabled = true;}
        };

        typedef list<Button *> Btns_t;
        typedef Btns_t::iterator BtnIter_t;
        Btns_t buttons;


        BtnIter_t currentBtn;

        int r, g, b,
                vpWidth, vpHeight,
                curFrame, lastFrame;

public:
        static void mousePassiveMotion(void *buttonsInstance, int x, int y);

        Buttons(int viewportWidth, int viewportHeight);
        ~Buttons();
        void display();
        void add(void *classInstance, int actionArg,
                void (*action)(void *classInstance, int actionArg),
                const string btnFiles[3], int xpos = 0, int ypos = 0);
        void unhoverAll();
        void unpressAll();
        void enable(int actionArg);
        void disable(int actionArg);
        bool areAllDisabled();


        void hoverPrev();
        void hoverNext();
        void pressCurrent();
        void unpressCurrent();
        void activateCurrent();


        Button * getBtnUnderCursor(int x, int y);
        void clickDown(int x, int y);
        void clickUp(int x, int y);
};


class Bonus : public CObject {
public:
        typedef enum { CHAIN, COMBO } BonusType;
        static const uint64 moveInterval;
        static const int moveSpeed, totalMove;
        CObject text;

        BonusType bonusType;
        int count;
        int row, col;
        uint64 lastMove;
        int offset;

public:
        Bonus(const Cell &cell, int cnt, BonusType bt, Grid &g);
        Bonus(const Bonus &);
        void set(const Cell &cell, int cnt, BonusType bt, Grid &g);
        void clone(const Bonus &);
        virtual ~Bonus();

        bool update();
        void display();
};

class FallNode : public Cell {
public:
        uint64 lastFall;
        int numFalls;
        bool enabled;
        int chainCount;

public:
        FallNode();
        FallNode(int r, int c);
        FallNode(const FallNode &src);
        FallNode & operator =(const FallNode &src);
        void clone(const FallNode &src);
        void set();

        void init(Grid &grid, int chains);
        void cleanup(Grid &grid);
        bool update(Grid &grid);

        int getChainCount() const { return chainCount; };
};

class Fall : public list<FallNode> {
protected:
        int chainCount;

public:
        static int fallInterval;

        Fall(int chains = 1);
        Fall(const Cell &, int chains = 0);
        Fall(const list<FallNode> &, int chains = 1);
        void set(int chains);
        void clone(const Fall &src);
        Fall & operator =(const Fall &src);

        void adjustRow();
        bool isEnabled();
        void enable();

        void init(Grid &grid);
        void cleanup(Grid &grid);
        bool update(Grid &grid);

        int getChainCount() const { return chainCount; }
};

class Block : public CObject {
public: enum gameState { inactive, enabled, disabled, combo, fall };
protected:
        gameState state;
        int fallOffset;
        uint64 lastFrameTransition;
        static const uint64 frameTransitionInterval;
        int currentFrame;

public:
        Block();
        Block(const Block &block);
        Block & operator =(const Block &block);
        void set();
        void clone(const Block &src);
        ~Block() {}

        friend bool swap(Block &left, Block &right);
        friend bool match(const Block &left, const Block &right, bool ignoreActive = false);

        void display();
        void composeFrame();
        void changeState(gameState gs);
        gameState getState() const { return state; }
        void fallDown();
        void resetFall();
        int getFallOffset() const;
        void setFallOffset(int f);

        static const int fallFactor;
};

class GamePlay {
public:
        enum gameState {play, pause, quit};
        static int blockLength;
        static int gridHeight;
        static int gridWidth;
        static void *font1;
        static float fcolor1[3], fcolor2[3];
        static const string bgFile, gridBorderFile, menuBarFile, bonusFile;
        static CBaseSprite *blockSprites[nblocktypes], *cursorSprite, *gridBorderSprite,
                *menuBarSprite, *bonusSprite, *chainFontSprite, *comboFontSprite, *gameOverSprite;

        Buttons * menuButtons;
        Grid *grid;

protected:
        static const int numCursorFiles;

        static BMPClass background;

        gameState state;
        HUD *hud;

public:
        static void declareActions();
        void defineActions();
        static void doAction(void *gridInstance, int actionState, int actionType);

        GamePlay();
        ~GamePlay();
        void display();
        void composeFrame();
        void init();
        void loadImages();

        static int getWidth() { return background.getViewportWidth();}
        static int getHeight() { return background.getViewportHeight();}

        void changeState(gameState gs);
        static void changeStateWrapper(void *gamePlayInstance, int state);
        gameState getState() const;
};

class Grid {

public:
        enum Difficulty {EASY, MEDIUM, HARD, YODA, NUMDIFFICULTIES};
        Cursor *cursor;
        deque< vector<Block> > blocks;
        enum gameState { play, combo, push, pause, gameover, quit };
```

```
protected:
        static const int forcedPushinterval;
        static const int startPushIntervals[NUMDIFFICULTIES];
        static const int endPushIntervals[NUMDIFFICULTIES];
        static const int pushAccelInterval;
        static const int gameOverDuration;
        int pushAccelDelta;

        int     pushOffset, pushInterval,
                comboInterval,
                current_cursor_frame;
        float pushAccel;
        uint64 last_cursor_anim, timer_cursor_anim;
        uint64 lastPush, lastPushAccel, lastForcedPush;
        uint64 startGameOver;
        Point gridPos;
        CBaseSprite** blockSprites;
        gameState state;
        Difficulty difficulty;

        list<Combo> comboEvents;
        list<Fall> fallEvents;
        list<Bonus> bonuses;

public:
        Grid();
        Grid(const Grid &g);
        void set();
        void init();
        void clone(const Grid &g);
        Grid & operator =(const Grid &g);
        virtual ~Grid();

        static void declareActions();
        void defineActions();
        static void doAction(void *gridInstance, int actionState, int actionType);

        void loadImages();
        void composeFrame();
        void display();
        void displayBonus();
        void updateEvents();

        void pushRow();
        void addRow();
        void swapBlocks();

        void changeState(gameState gs);
        gameState getState() const { return state; }

        void setDifficulty(Difficulty diff);
        Difficulty getDifficulty() const { return difficulty; }

        /* set/get properties */
        int getYOffset() { return pushOffset; }
        int getX() { return gridPos.x; }
        int getY() { return gridPos.y; }
        int countEnabledRows() const;
```

```
	int downMatch(int r, int c, bool ignoreActive = false);
	int upMatch(int r, int c, bool ignoreActive = false);
	int leftMatch(int r, int c, bool ignoreActive = false);
	int rightMatch(int r, int c, bool ignoreActive = false);

	void incComboInterval(int interval);
	bool containsPoint(int x, int y);
	bool containsPoint(Point point);

	bool detectFall(const Combo & combo);
	bool detectFall(int r, int c, bool initialize = true);

	Combo detectCombo(Cell &cell, int chains, bool initialize);
	void initCombo(Combo &combo);
	void initBonus(const Combo &combo);

	void setBlockStates(list<Cell> &, Block::gameState gs);

	void printDebug();
};

class Cursor : public CObject {
protected:
	int row, col,
		cursor_delta;
	Point lastMousePos;
	Grid *grid;
	static const string spriteFile;


	static uint64 cursorMoveDelay, cursorMoveInitialDelay;
	uint64 cursorMoveInitial[PlayState::_NUMBER_OF_ACTIONS],
		   cursorMoveLast[PlayState::_NUMBER_OF_ACTIONS];

public:
	static void declareActions();
	void defineActions();
	void doActionPress(PlayState::Actions action);
	void doActionHold(PlayState::Actions action);
	static void doAction(void *cursorInstance, int actionState, int actionType);
	void alignToMouse();
	bool isAlignedToMouse;
	static void mousePassiveMotion(void *cursorInstance, int x, int y);

	Cursor(Grid *, CBaseSprite *);
	~Cursor();
	bool move(PlayState::Actions action, bool doDraw=true);
	bool moveUp(bool doDraw=true);
	bool moveDown(bool doDraw=true);
	bool moveLeft(bool doDraw=true);
	bool moveRight(bool doDraw=true);
	void setPos(int c, int r);
	int getRow() const { return row; }
	int getCol() const { return col; }
	void shiftRow();

	Point getMousePos() {return lastMousePos;}
};

class HUD {
```

```
protected:
	uint64 startTime;
	uint64 currentTime;
	int currentSeconds;
	int currentMinutes;
	float x, y;
	char timeStr[10];

protected:
	void calcTime();
	void init(int x, int y);

public:
	HUD(int x, int y);
	void composeFrame();
	void display();
}

namespace Input
{

	class Action
	{
	public:
		enum ActionState {STATE_PRESS, STATE_HOLD, STATE_RELEASE};
		enum ActionScope {SCOPE_FIRST_PLAYER, SCOPE_CURRENT_PLAYER,
SCOPE_ALL_PLAYERS};
		typedef void (*ActionFunc)(void *actionsClassInstance,
											int actionState, int actionType);
	private:
		ActionScope scope;
		int activeState;
		int actionType;
		string shortDesc;

		void *actionsClassInstance;
		ActionFunc action;
	public:
		Action() : action(NULL), actionsClassInstance(NULL){};

		Action(ActionScope scope, int activeState, int actionType, string shortDesc) :
		  action(NULL), actionsClassInstance(NULL),
		  scope(scope), activeState(activeState), actionType(actionType),
		  shortDesc(shortDesc){}

		bool isDefined() {return actionsClassInstance && action;}
		bool isSameAction(Action * action);
		bool isSameAction(ActionScope scope, int activeState, int actionType);
		bool isRelatedAction(ActionScope scope, int activeState);
		bool isFor(void *classInstance) {return classInstance == this->actionsClassInstance;}
		bool hasActiveStateOf(int activeState) {return this->activeState == activeState;}

		void doAction(int actionState);
	   void define(void *actionsClassInstance, ActionFunc);
		void undefine();
	};

	class Player
	{
	private:
		bool enabled;
		list<Action *> actions;
	public:
		Player();
		void addAction(Action *action);
		bool isActionDefined(Action::ActionScope scope, int activeState, int actionType);
		bool hasActionsDefined(Action::ActionScope scope, int activeState);
		Action *getAction(Action::ActionScope scope, int activeState, int actionType);
		void undefineActions(void *classInstance);
		void enable();
		void disable();
		bool isEnabled() {return enabled;}
	};

}
```