<^> TRIFORCE <^>

Design Workflow

by Gabe Pike

CMPS 335

Spring 2012

Team 10

Other Team Member: Brandon Hinesley

Table of Contents

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

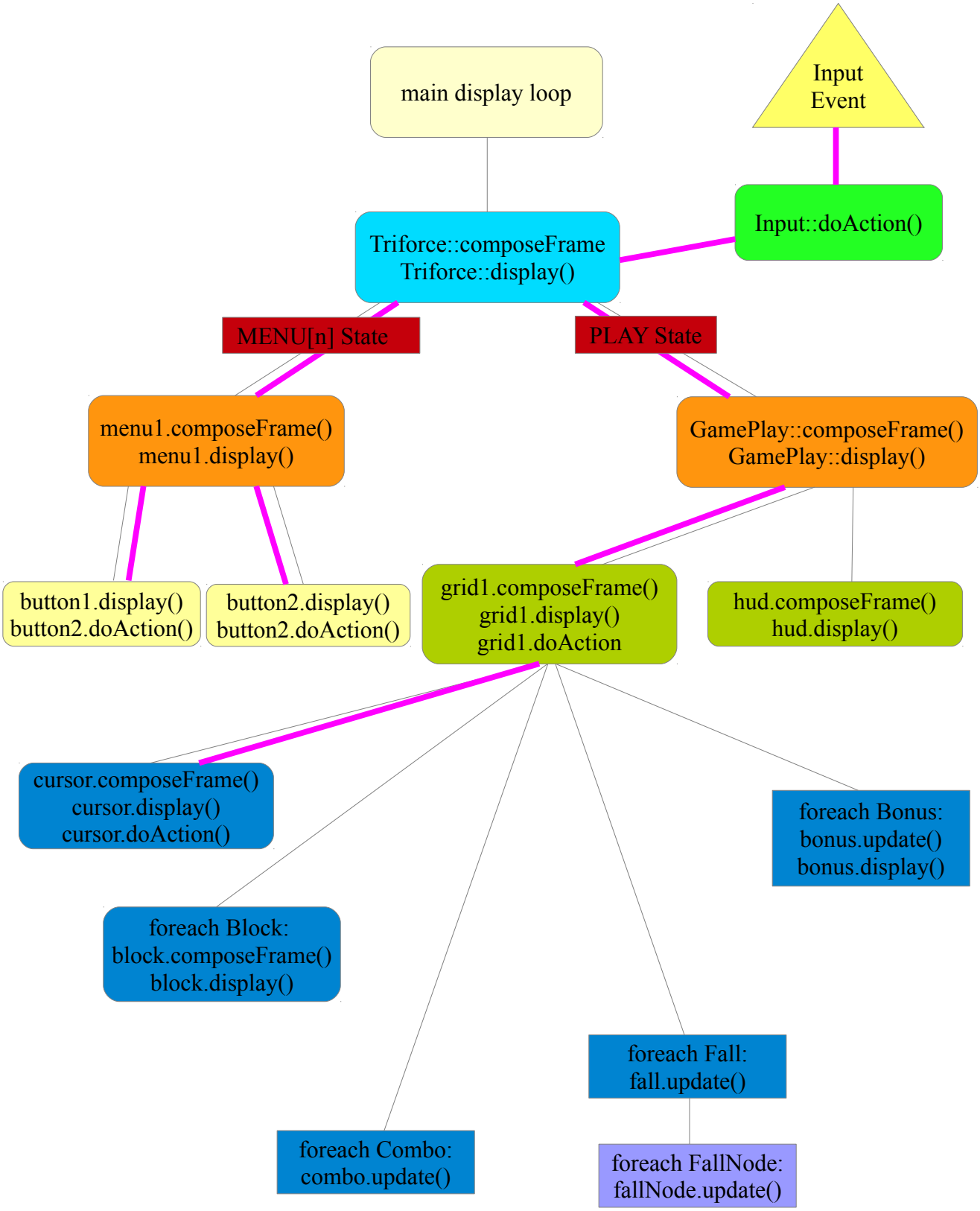
3.1.3 Architectural Design

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 Combo::Combo Combo::changeState 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 Combo::getState 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 Combo::getState 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 Block::offsetY cursor::shiftRow 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: Load the main menu Control flow of the program based on the current state Set up input bindings/handlers
Collaboration: GamePlay Input

Class: Input
Responsibility: Provide an interface to bind input events to actions (callback methods)
Collaboration: Triforce GamePlay Button Grid Cursor

Class: GamePlay
Responsibility: Initialize the game Handle input that affects the state of GamePlay Control the flow among objects in GamePlay Initialize objects that will be displayed during the game
Collaboration: Grid Input 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

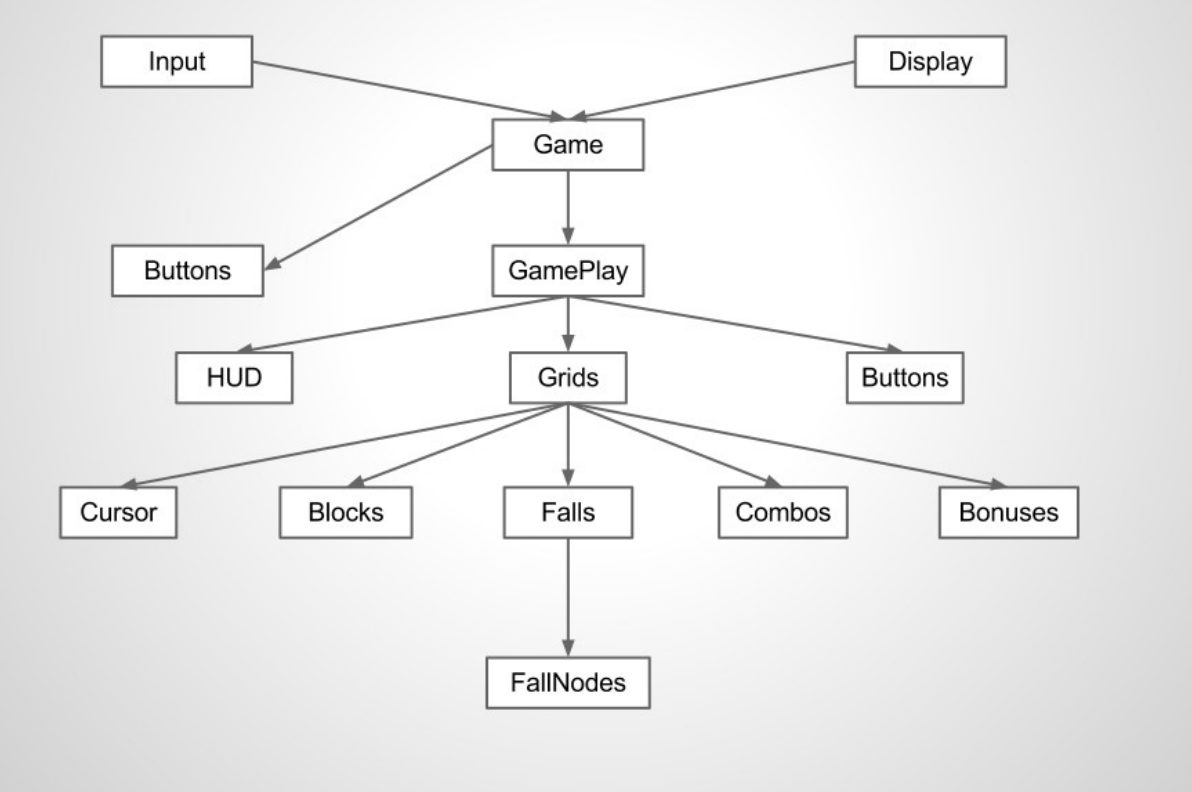
3.3.3 Architectural Design

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 Combo::changeState Combo::init 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 Combo::getState 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 Combo::getState 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 Block::offsetY cursor::shiftRow 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();
        }
}
```