Toddtris - UML Model

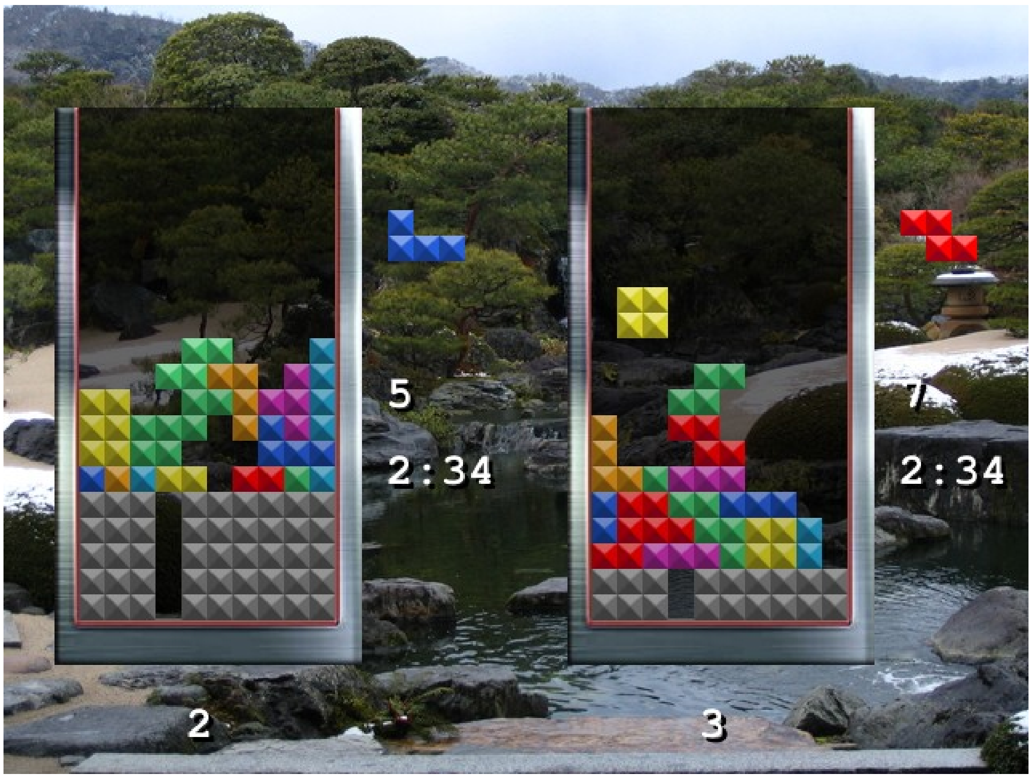

Table of Contents

1 Model

The 'model' namespace includes the classes that model the physical state of the game. The model classes have been designed to minimise object creation and memory usage owing to the code's origin in a Java ME game.

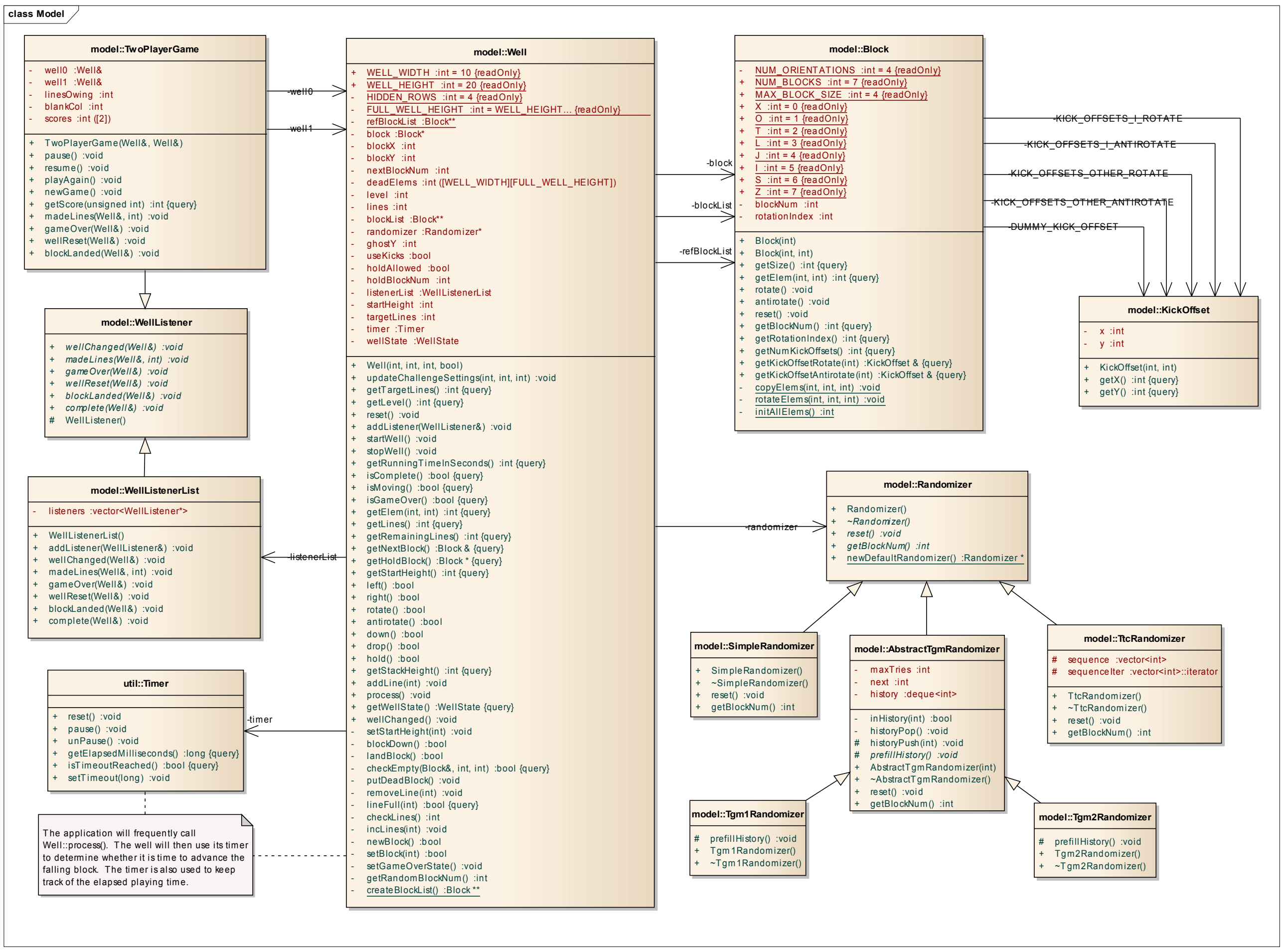

Illustration 1: Toddtris model classes

1.1 Block

The Block class represents one of the seven possible game pieces and includes methods to rotate the block. A block is represented as a square composed of 2x2, 3x3 or 4x4 elements. An element is either (a) 0 representing empty space or (b) a non-zero number representing part of the game piece. The 'size' of a block is found using the 'getSize' method and the value of a particular element is found using 'getElem'.

In practice all four possible orientations of each of the seven different blocks are pre-calculated and the state of the Block instance is determined by 'blockNum' denoting which of the seven block types this block represents and 'rotationIndex' denoting its orientation. A 'reset' method is provided to return the block to its original orientation to assist in reusing Block instances.

1.2 Well

'Well' is the most important class in the toddtris game. This class represents the well into which pieces are dropped, the blocks that have already landed, the block currently in play, the 'next' block and the 'ghost' block. The 'ghost' block shows the position in which the block would land if dropped from current position.

Methods are included to rotate and move the in-play block, display or hide the 'ghost' block, add lines to the bottom of the Well for use in two player games and to store or use the 'hold' block (note that the hold block facility is disabled by not being connected to any input as the author does not like this facility).

The arrangement of elements in the well is implemented by using a two dimensional array to denoting the 'landed' elements and the (x, y) coordinates of the in-play block. The 'getElem' method overlays the in-play block onto the well to give a snapshot view of the state of the current state of the well.

Description of element	Calculated as	Possible values
Empty space	0	0
Block in play	Block Number	1 – 7
Landed block	0 – (Block Number)	(-1) – (-7)
'Dead element' (added during two player game)	Block::NUM_BLOCKS + 1	8
Ghost element	Block Number + Block::NUM_BLOCKS + 1	9 – 15

1.3 *Randomizer*

Randomizer is a pure virtual class is used by the Well class to select blocks at random. Several different algorithms are implemented.

1.4 *KickOffset*

The KickOffset class is used in the implementation of 'wall kicks'. If an attempt to rotate a block fails (i.e. the rotated block does not fit at its current coordinates) then a series of small displacements of the block are attempted. For more details see `http://harddrop.com/wiki/SRS` .

1.5 *TwoPlayerGame*

The TwoPlayerGame class is used to model a two player battle. Lines made by each player are tracked and result in lines being added to the bottom of the opposing player's well. The number of games won by each player is also tracked.

1.6 *WellListener / WellListenerList*

WellListener is a pure virtual class which may be extended by classes wishing to be notified of changes to the state of a well through various callback methods. For example the TwoPlayerGame class extends WellListener so that it can be notified of lines made by each player adding lines to the other player's well in response. The Well::addListener method is used to register listeners.

WellListenerList the class used by Well to keep track of registered listeners.

2 State Machine

A state machine pattern is used to handle the different logical states the game may be – in menu system, playing one player game, playing two player game, paused, game over, etc. Depending on the current state input will be handled differently and may cause transition to another state. The states of the application are shown below.

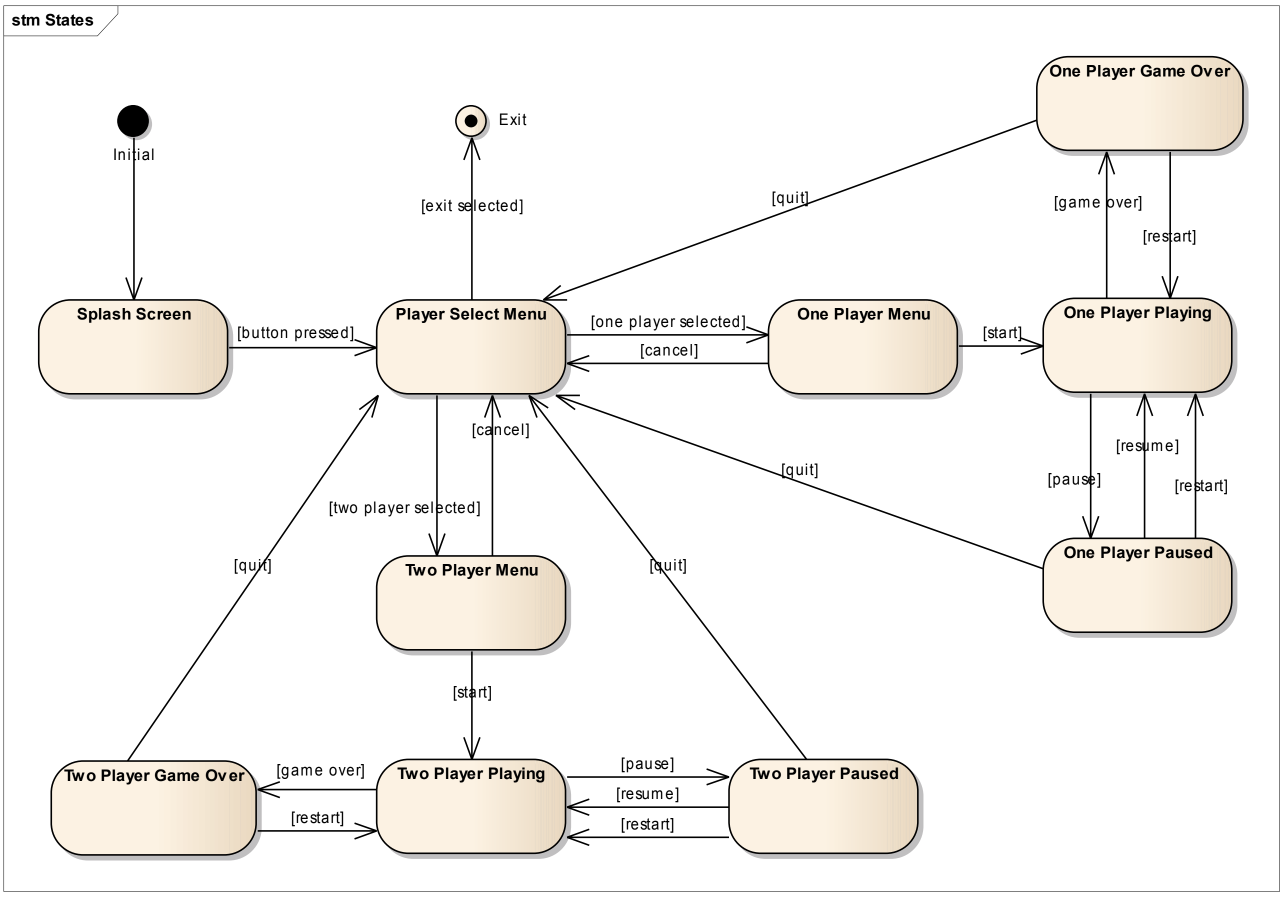

Illustration 2: Toddtris application states

2.1 Implementation of State Machine

The StateMachine is implemented by classes in the “app” namespace.

2.2 AppState

The AppState class is the superclass for all of the games' states. Empty virtual methods exist which are invoked by the state machine framework in response to button presses from players. Any state interested in a particular button press need simply override the relevant method to receive callbacks. A virtual 'prepare' method may also be overridden by a state to receive a callback that this state has been entered so that it might carry out any required set up; for example when a paused state is entered it will stop the timer used by the well to automatically drop blocks.

As a simple approximation to multi-threading a 'process' method may be overridden to perform any background processing and will be called frequently by the state machine framework. Any implementation of 'process' should quickly return control to the framework by simply returning from this method. In practice this method is overridden in states where the game is in progress to allow blocks to fall by checking a timer and moving the blocks down when required.

A pure virtual 'draw' method must be implemented by all states. This method is responsible for rendering the current state on screen and will be invoked by the state machine framework.

When an event occurs giving raise to a state transition the state will notify the state machine framework via a callback to the AppStateMachine class (see next section). The framework will then be responsible for handling the transition. For example, when the player starts a new one player game, the OnePlayerMenuState class will trigger a state transition by calling

AppStateMachine::startOnePlayerGame.

2.3 AppStateMachine

AppStateMachine is responsible for reading the input state (i.e. button presses) and calling the relevant methods on the current state in response to those button presses. AppStateMachine will also call the 'process' method on each state after reading the input as described in the previous section and then request that the state render itself. This process is repeated in an loop until the 'exitApp' method is called. A number of methods are provided which the current state may use to invoke state transitions.

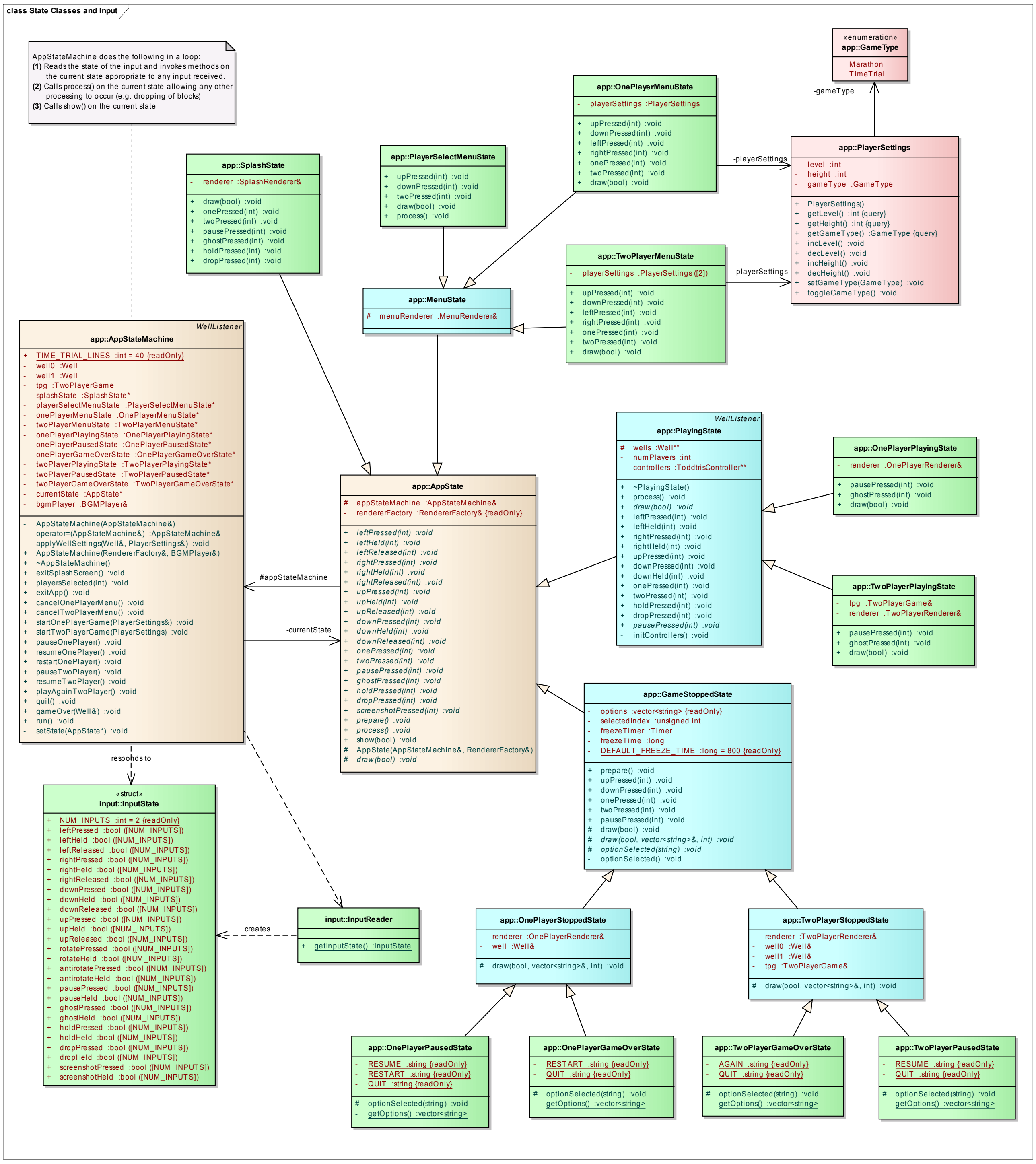
class State Classes and Input
AppStateMachine does the following in a loop:
(1) Reads the state of the input and invokes methods on the current state appropriate to any input received.
(2) Calls process() on the current state allowing any other processing to occur (e.g. dropping of blocks)
(3) Calls show() on the current state
app::SplashState
- renderer :SplashRenderer&
+ draw(bool) :void
+ onePressed(int) :void
+ twoPressed(int) :void
+ pausePressed(int) :void
+ ghostPressed(int) :void
+ holdPressed(int) :void
+ dropPressed(int) :void
app::PlayerSelectMenuState
+ upPressed(int) :void
+ downPressed(int) :void
+ twoPressed(int) :void
+ draw(bool) :void
+ process() :void
app::OnePlayerMenuState
- playerSettings :PlayerSettings
+ upPressed(int) :void
+ downPressed(int) :void
+ leftPressed(int) :void
+ rightPressed(int) :void
+ onePressed(int) :void
+ twoPressed(int) :void
+ draw(bool) :void
«enumeration»
app::GameType
Marathon
TimeTrial
-gameType
-playerSettings
app::PlayerSettings
- level :int
- height :int
- gameType :GameType
+ PlayerSettings()
+ getLevel() :int {query}
+ getHeight() :int {query}
+ getGameType() :GameType {query}
+ incLevel() :void
+ decLevel() :void
+ incHeight() :void
+ decHeight() :void
+ setGameType(GameType) :void
+ toggleGameType() :void
app::MenuState
# menuRenderer :MenuRenderer&
app::TwoPlayerMenuState
- playerSettings :PlayerSettings ([2])
+ upPressed(int) :void
+ downPressed(int) :void
+ leftPressed(int) :void
+ rightPressed(int) :void
+ onePressed(int) :void
+ twoPressed(int) :void
+ draw(bool) :void
-playerSettings
WellListener
app::AppStateMachine
+ TIME_TRIAL_LINES :int = 40 {readOnly}
- well0 :Well
- well1 :Well
- tpg :TwoPlayerGame
- splashState :SplashState*
- playerSelectMenuState :PlayerSelectMenuState*
- onePlayerMenuState :OnePlayerMenuState*
- twoPlayerMenuState :TwoPlayerMenuState*
- onePlayerPlayingState :OnePlayerPlayingState*
- onePlayerPausedState :OnePlayerPausedState*
- onePlayerGameOverState :OnePlayerGameOverState*
- twoPlayerPlayingState :TwoPlayerPlayingState*
- twoPlayerPausedState :TwoPlayerPausedState*
- twoPlayerGameOverState :TwoPlayerGameOverState*
- currentState :AppState*
- bgmPlayer :BGMPlayer&
- AppStateMachine(AppStateMachine&)
- operator=(AppStateMachine&) :AppStateMachine&
- applyWellSettings(Well&, PlayerSettings&) :void
+ AppStateMachine(RendererFactory&, BGMPlayer&)
+ ~AppStateMachine()
+ exitSplashScreen() :void
+ playersSelected(int) :void
+ exitApp() :void
+ cancelOnePlayerMenu() :void
+ cancelTwoPlayerMenu() :void
+ startOnePlayerGame(PlayerSettings&) :void
+ startTwoPlayerGame(PlayerSettings) :void
+ pauseOnePlayer() :void
+ resumeOnePlayer() :void
+ restartOnePlayer() :void
+ pauseTwoPlayer() :void
+ resumeTwoPlayer() :void
+ playAgainTwoPlayer() :void
+ quit() :void
+ gameOver(Well&) :void
+ run() :void
- setState(AppState*) :void
#appStateMachine
-currentState
app::AppState
# appStateMachine :AppStateMachine&
- rendererFactory :RendererFactory& {readOnly}
+ leftPressed(int) :void
+ leftHeld(int) :void
+ leftReleased(int) :void
+ rightPressed(int) :void
+ rightHeld(int) :void
+ rightReleased(int) :void
+ upPressed(int) :void
+ upHeld(int) :void
+ upReleased(int) :void
+ downPressed(int) :void
+ downHeld(int) :void
+ downReleased(int) :void
+ onePressed(int) :void
+ twoPressed(int) :void
+ pausePressed(int) :void
+ ghostPressed(int) :void
+ holdPressed(int) :void
+ dropPressed(int) :void
+ screenshotPressed(int) :void
+ prepare() :void
+ process() :void
+ show(bool) :void
# AppState(AppStateMachine&, RendererFactory&)
# draw(bool) :void
WellListener
app::PlayingState
# wells :Well**
- numPlayers :int
- controllers :ToddtrisController**
+ ~PlayingState()
+ process() :void
+ draw(bool) :void
+ leftPressed(int) :void
+ leftHeld(int) :void
+ rightPressed(int) :void
+ rightHeld(int) :void
+ upPressed(int) :void
+ downPressed(int) :void
+ downHeld(int) :void
+ onePressed(int) :void
+ twoPressed(int) :void
+ holdPressed(int) :void
+ dropPressed(int) :void
+ pausePressed(int) :void
- initControllers() :void
app::OnePlayerPlayingState
- renderer :OnePlayerRenderer&
+ pausePressed(int) :void
+ ghostPressed(int) :void
+ draw(bool) :void
app::TwoPlayerPlayingState
- tpg :TwoPlayerGame&
- renderer :TwoPlayerRenderer&
+ pausePressed(int) :void
+ ghostPressed(int) :void
+ draw(bool) :void
app::GameStoppedState
- options :vector<string> {readOnly}
- selectedIndex :unsigned int
- freezeTimer :Timer
- freezeTime :long
- DEFAULT_FREEZE_TIME :long = 800 {readOnly}
+ prepare() :void
+ upPressed(int) :void
+ downPressed(int) :void
+ onePressed(int) :void
+ twoPressed(int) :void
+ pausePressed(int) :void
# draw(bool) :void
# draw(bool, vector<string>&, int) :void
# optionSelected(string) :void
- optionSelected() :void
responds to
«struct»
input::InputState
+ NUM_INPUTS :int = 2 {readOnly}
+ leftPressed :bool ([NUM_INPUTS])
+ leftHeld :bool ([NUM_INPUTS])
+ leftReleased :bool ([NUM_INPUTS])
+ rightPressed :bool ([NUM_INPUTS])
+ rightHeld :bool ([NUM_INPUTS])
+ rightReleased :bool ([NUM_INPUTS])
+ downPressed :bool ([NUM_INPUTS])
+ downHeld :bool ([NUM_INPUTS])
+ downReleased :bool ([NUM_INPUTS])
+ upPressed :bool ([NUM_INPUTS])
+ upHeld :bool ([NUM_INPUTS])
+ upReleased :bool ([NUM_INPUTS])
+ rotatePressed :bool ([NUM_INPUTS])
+ rotateHeld :bool ([NUM_INPUTS])
+ antirotatePressed :bool ([NUM_INPUTS])
+ antirotateHeld :bool ([NUM_INPUTS])
+ pausePressed :bool ([NUM_INPUTS])
+ pauseHeld :bool ([NUM_INPUTS])
+ ghostPressed :bool ([NUM_INPUTS])
+ ghostHeld :bool ([NUM_INPUTS])
+ holdPressed :bool ([NUM_INPUTS])
+ holdHeld :bool ([NUM_INPUTS])
+ dropPressed :bool ([NUM_INPUTS])
+ dropHeld :bool ([NUM_INPUTS])
+ screenshotPressed :bool ([NUM_INPUTS])
+ screenshotHeld :bool ([NUM_INPUTS])
creates
input::InputReader
+ getInputState() :InputState
app::OnePlayerStoppedState
- renderer :OnePlayerRenderer&
- well :Well&
# draw(bool, vector<string>&, int) :void
app::TwoPlayerStoppedState
- renderer :TwoPlayerRenderer&
- well0 :Well&
- well1 :Well&
- tpg :TwoPlayerGame&
# draw(bool, vector<string>&, int) :void
app::OnePlayerPausedState
- RESUME :string {readOnly}
- RESTART :string {readOnly}
- QUIT :string {readOnly}
# optionSelected(string) :void
- getOptions() :vector<string>
app::OnePlayerGameOverState
- RESTART :string {readOnly}
- QUIT :string {readOnly}
# optionSelected(string) :void
- getOptions() :vector<string>
app::TwoPlayerGameOverState
- AGAIN :string {readOnly}
- QUIT :string {readOnly}
# optionSelected(string) :void
- getOptions() :vector<string>
app::TwoPlayerPausedState
- RESUME :string {readOnly}
- QUIT :string {readOnly}
# optionSelected(string) :void
- getOptions() :vector<string>

Illustration 3: Toddtris state machine implementation

2.4 InputReader / InputState

These classes are used by AppStateMachine to read button presses. Different implementations exist for different platforms.

2.5 main.cpp / init.cpp

main.cpp provides the 'main' method for the game. It simply does some initialisation and then hands control to the state machine. The initialisation is handled by the 'init' method in init.cpp of which different versions exist depending on the (a) the platform and (b) the renderer to be used (text-based or graphical).

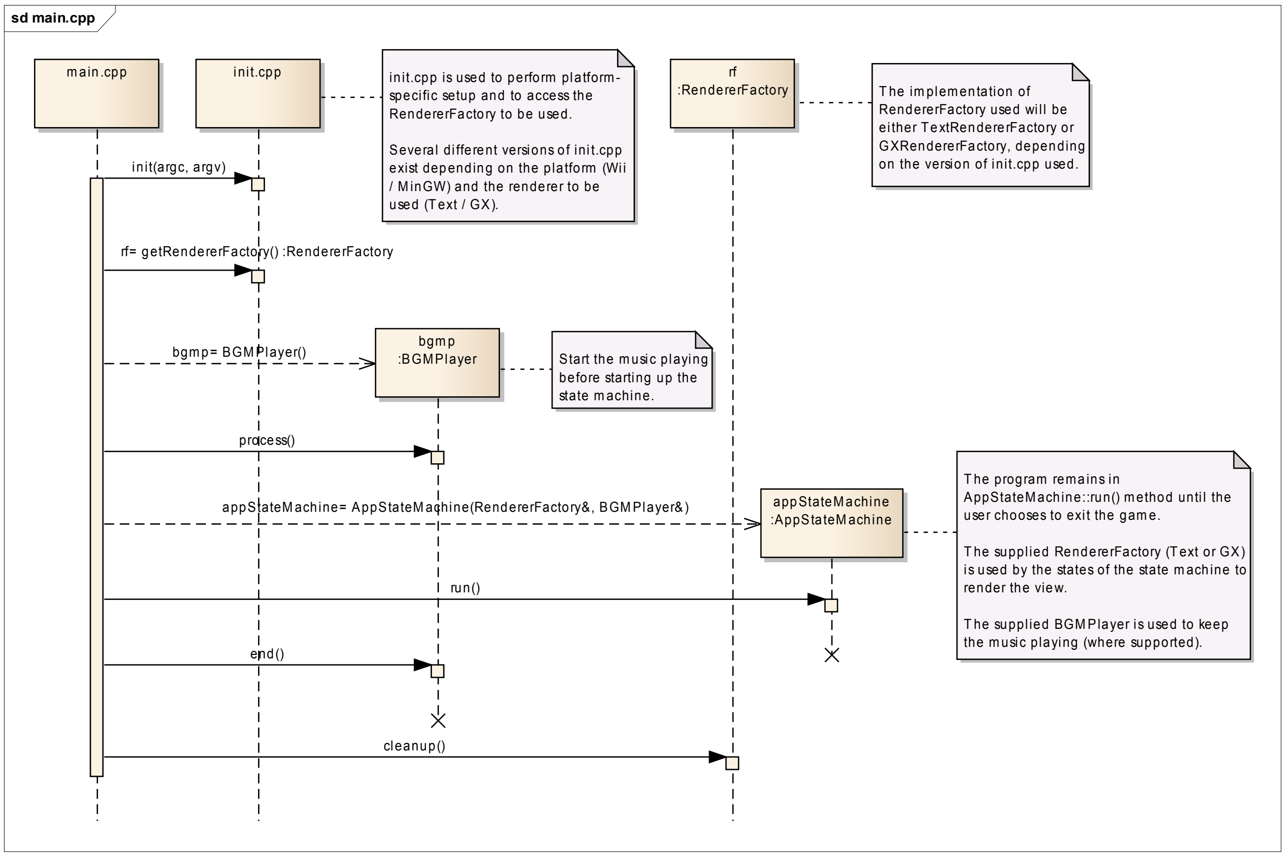
sd main.cpp
main.cpp
init.cpp
init.cpp is used to perform platform-specific setup and to access the RendererFactory to be used.
Several different versions of init.cpp exist depending on the platform (Wii / MinGW) and the renderer to be used (Text / GX).
rf :RendererFactory
The implementation of RendererFactory used will be either TextRendererFactory or GXRendererFactory, depending on the version of init.cpp used.
init(argc, argv)
rf= getRendererFactory() :RendererFactory
bgmp :BGMPlayer
bgmp= BGMPlayer()
Start the music playing before starting up the state machine.
process()
appStateMachine= AppStateMachine(RendererFactory&, BGMPlayer&)
appStateMachine :AppStateMachine
The program remains in AppStateMachine::run() method until the user chooses to exit the game.
The supplied RendererFactory (Text or GX) is used by the states of the state machine to render the view.
The supplied BGMPlayer is used to keep the music playing (where supported).
run()
end()
cleanup()

Illustration 4: Outline of Toddtris state machine process

3 Well Controller

The WellController class is used when the game is in a playing state to manipulate the players' wells.

A WellController is created for each well and the PlayingState class delegates button presses to this. WellController will in turn delegate left, right and down button presses to 'repeater' classes. The repeater classes are used to repeat a movement when the player holds a button down.

The WellController and repeaters further delegate to an ActionHandler subclass which in turn manipulates the well. The point of the ActionHandler is that there is StillActionHandler for use when the game is paused and a MovingActionHandler for use when the game is in progress. In practice this is not really required; In the current code base WellController is only ever used when the game is in a 'playing' state.

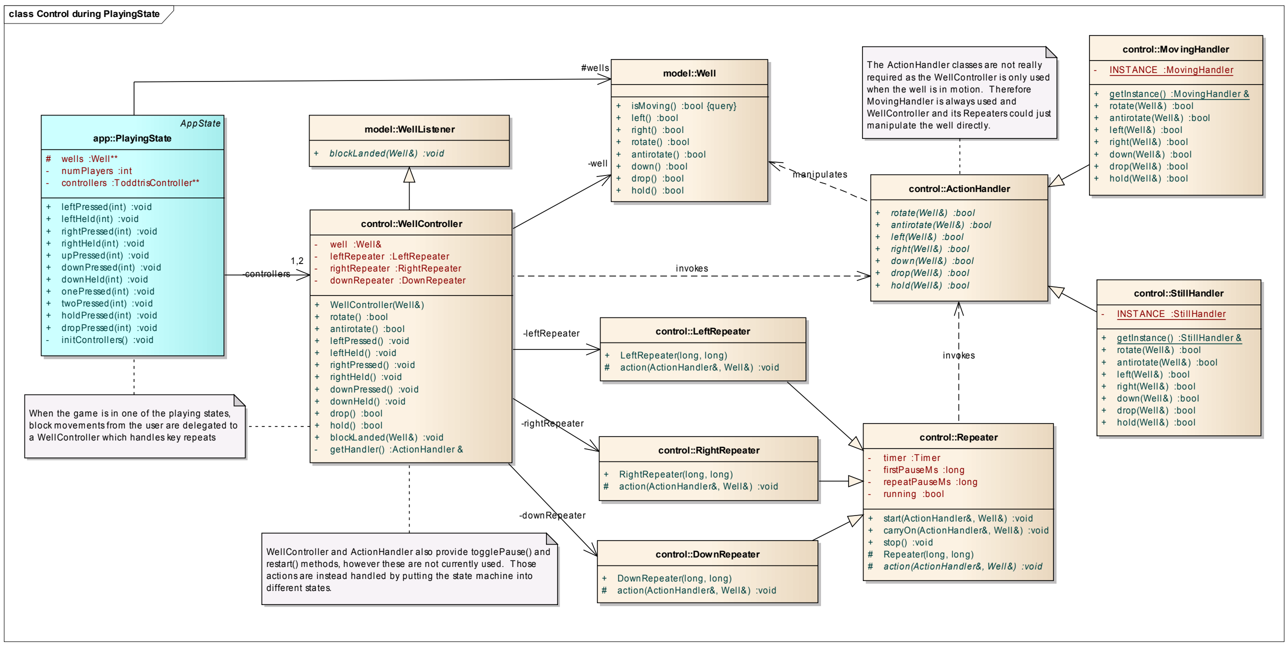

Illustration 5: Toddtris WellController class

4 Rendering

Rendering of the game on screen is handled by various 'renderer' classes which are called by the state classes (Section 2) to render the current state on screen. The state classes deal with abstract renderer classes. Both text-based and graphical implementations of the renderers are available which may be swapped in by simply changing the RendererFactory implementation which is passed to the state machine on startup.

4.1 SplashRenderer

Superclass for simple renderers which are required simply to render the splash screen which is shown when the game starts.

4.2 MenuRenderer

MenuRenderer is used by the various menu states to render menu screens with highlighting of selected options.

4.3 OnePlayerRenderer

Used to draw the screen for a one player game that may be in the in-play, paused or game over state.

4.4 TwoPlayerRenderer

Used to draw the screen for a two player game that may be in the in-play, paused or game over state.

4.5 WellRenderer

WellRenderer is used by OnePlayerRenderer and TwoPlayerRenderer above to render wells and associated detail on screen including the next block, time, and lines made.

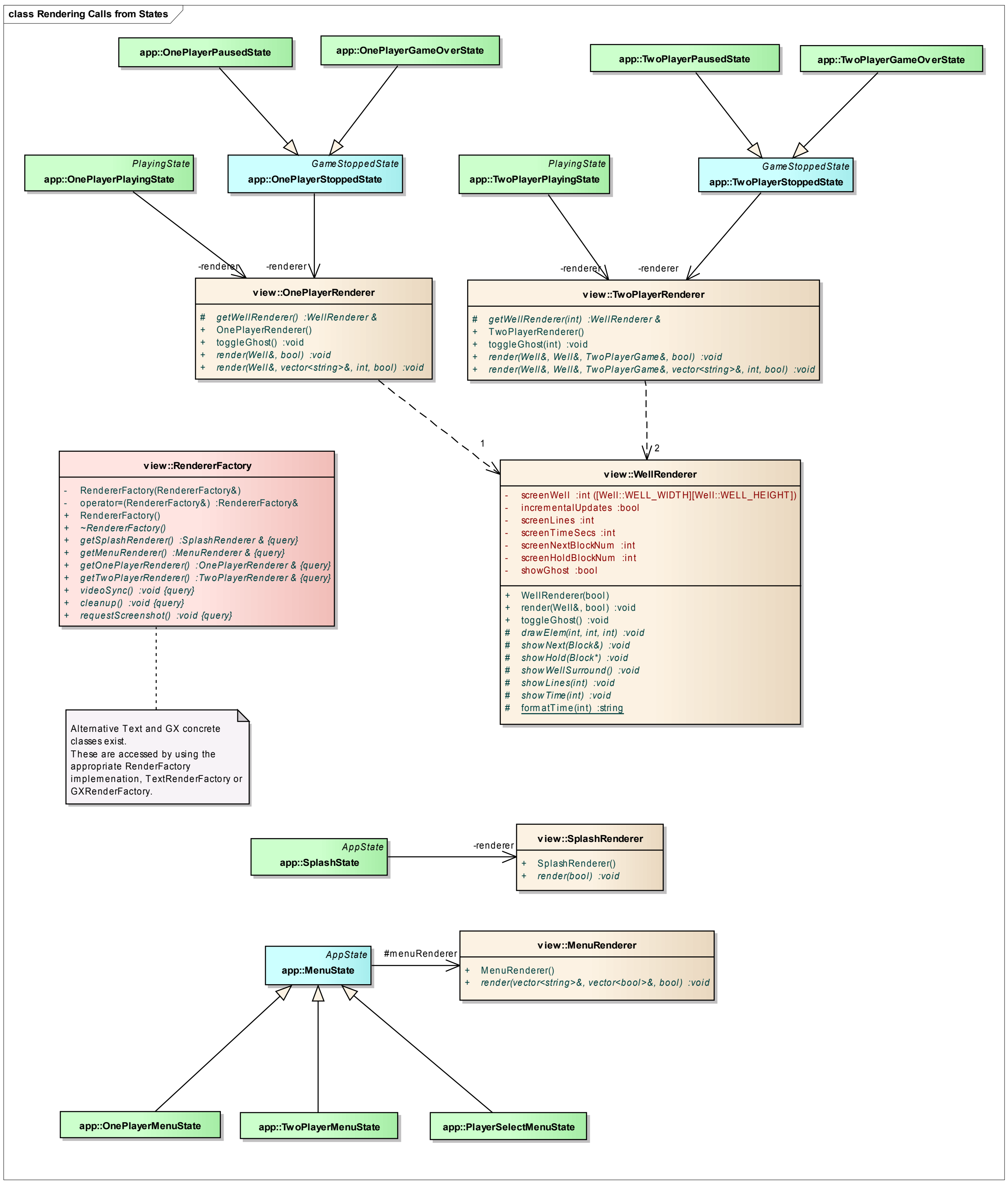
class Rendering Calls from States
app::OnePlayerPausedState
app::OnePlayerGameOverState
app::TwoPlayerPausedState
app::TwoPlayerGameOverState
PlayingState
app::OnePlayerPlayingState
GameStoppedState
app::OnePlayerStoppedState
PlayingState
app::TwoPlayerPlayingState
GameStoppedState
app::TwoPlayerStoppedState
-renderer
-renderer
-renderer
-renderer
view::OnePlayerRenderer
# getWellRenderer() :WellRenderer &
+ OnePlayerRenderer()
+ toggleGhost() :void
+ render(Well&, bool) :void
+ render(Well&, vector<string>&, int, bool) :void
view::TwoPlayerRenderer
# getWellRenderer(int) :WellRenderer &
+ TwoPlayerRenderer()
+ toggleGhost(int) :void
+ render(Well&, Well&, TwoPlayerGame&, bool) :void
+ render(Well&, Well&, TwoPlayerGame&, vector<string>&, int, bool) :void
1
2
view::RendererFactory
- RendererFactory(RendererFactory&)
- operator=(RendererFactory&) :RendererFactory&
+ RendererFactory()
+ ~RendererFactory()
+ getSplashRenderer() :SplashRenderer & {query}
+ getMenuRenderer() :MenuRenderer & {query}
+ getOnePlayerRenderer() :OnePlayerRenderer & {query}
+ getTwoPlayerRenderer() :TwoPlayerRenderer & {query}
+ videoSync() :void {query}
+ cleanup() :void {query}
+ requestScreenshot() :void {query}
view::WellRenderer
- screenWell :int ([Well::WELL_WIDTH][Well::WELL_HEIGHT])
- incrementalUpdates :bool
- screenLines :int
- screenTimeSecs :int
- screenNextBlockNum :int
- screenHoldBlockNum :int
- showGhost :bool
+ WellRenderer(bool)
+ render(Well&, bool) :void
+ toggleGhost() :void
# drawElem(int, int, int) :void
# showNext(Block&) :void
# showHold(Block*) :void
# showWellSurround() :void
# showLines(int) :void
# showTime(int) :void
# formatTime(int) :string
Alternative Text and GX concrete classes exist.
These are accessed by using the appropriate RenderFactory implemenation, TextRenderFactory or GXRenderFactory.
AppState
app::SplashState
-renderer
view::SplashRenderer
+ SplashRenderer()
+ render(bool) :void
AppState
app::MenuState
#menuRenderer
view::MenuRenderer
+ MenuRenderer()
+ render(vector<string>&, vector<bool>&, bool) :void
app::OnePlayerMenuState
app::TwoPlayerMenuState
app::PlayerSelectMenuState

Illustration 6: Use of renderers by state classes

4.6 Renderer Implementations

Two implementations of the renderer classes are available – text-based renderers or graphical renderers. The graphical renderers are available only for the Wii version of the game. Which renderers are used is determined simply by the implementation of RendererFactory which is passed to the state machine and used by the state classes to access concrete implementations of the abstract renderer classes defined above.

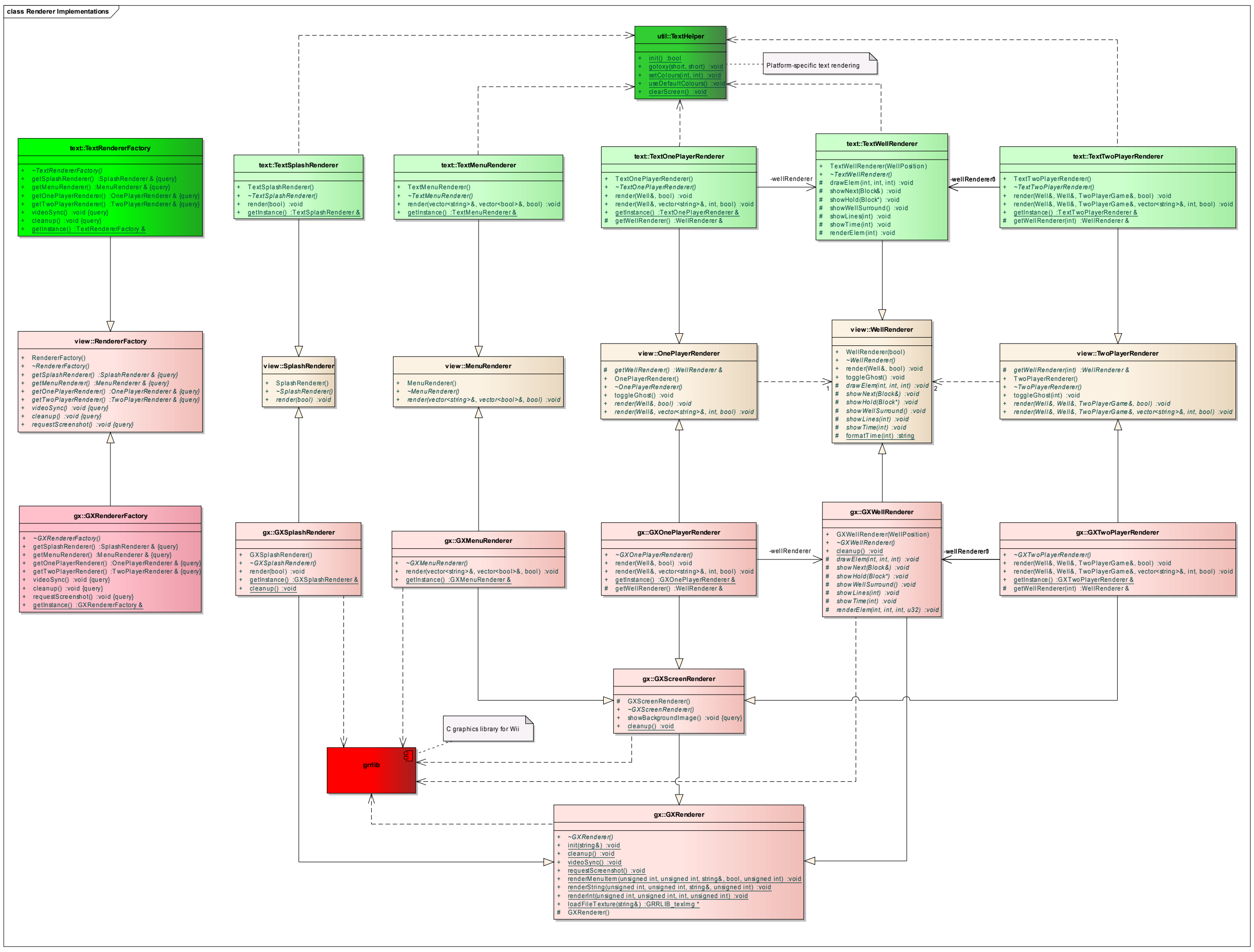

Illustration 7: Text based vs graphical implementation of renderer classes

5 Platform Specific Code

The method of reading input, rendering text on screen and determining the current system time varies between platforms. These functions are performed via helper classes which are implemented differently according to the platform. The affected classes are shown below. Note that the BGMPlayer implementation for MinGW is simply a dummy implementation and will not actually play the background music.

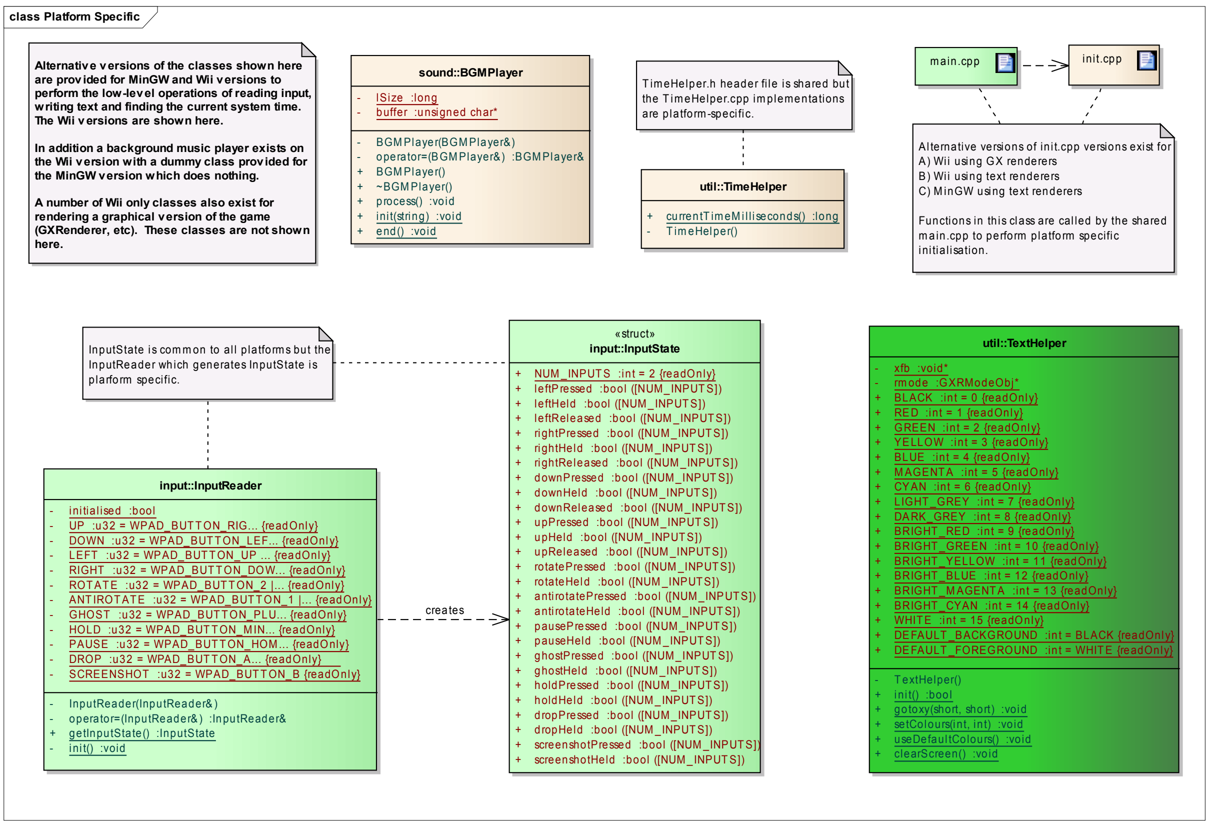
class Platform Specific
Alternative versions of the classes shown here are provided for MinGW and Wii versions to perform the low-level operations of reading input, writing text and finding the current system time. The Wii versions are shown here.
In addition a background music player exists on the Wii version with a dummy class provided for the MinGW version which does nothing.
A number of Wii only classes also exist for rendering a graphical version of the game (GXRenderer, etc). These classes are not shown here.
sound::BGMPlayer
- lSize :long
- buffer :unsigned char*
- BGMPlayer(BGMPlayer&)
- operator=(BGMPlayer&) :BGMPlayer&
+ BGMPlayer()
+ ~BGMPlayer()
+ process() :void
+ init(string) :void
+ end() :void
TimeHelper.h header file is shared but the TimeHelper.cpp implementations are platform-specific.
util::TimeHelper
+ currentTimeMilliseconds() :long
- TimeHelper()
main.cpp
init.cpp
Alternative versions of init.cpp versions exist for
A) Wii using GX renderers
B) Wii using text renderers
C) MinGW using text renderers
Functions in this class are called by the shared main.cpp to perform platform specific initialisation.
InputState is common to all platforms but the InputReader which generates InputState is plarform specific.
input::InputReader
- initialised :bool
- UP :u32 = WPAD_BUTTON_RIG... {readOnly}
- DOWN :u32 = WPAD_BUTTON_LEF... {readOnly}
- LEFT :u32 = WPAD_BUTTON_UP ... {readOnly}
- RIGHT :u32 = WPAD_BUTTON_DOW... {readOnly}
- ROTATE :u32 = WPAD_BUTTON_2 |... {readOnly}
- ANTIROTATE :u32 = WPAD_BUTTON_1 |... {readOnly}
- GHOST :u32 = WPAD_BUTTON_PLU... {readOnly}
- HOLD :u32 = WPAD_BUTTON_MIN... {readOnly}
- PAUSE :u32 = WPAD_BUTTON_HOM... {readOnly}
- DROP :u32 = WPAD_BUTTON_A... {readOnly}
- SCREENSHOT :u32 = WPAD_BUTTON_B {readOnly}
- InputReader(InputReader&)
- operator=(InputReader&) :InputReader&
+ getInputState() :InputState
- init() :void
creates
«struct»
input::InputState
+ NUM_INPUTS :int = 2 {readOnly}
+ leftPressed :bool ([NUM_INPUTS])
+ leftHeld :bool ([NUM_INPUTS])
+ leftReleased :bool ([NUM_INPUTS])
+ rightPressed :bool ([NUM_INPUTS])
+ rightHeld :bool ([NUM_INPUTS])
+ rightReleased :bool ([NUM_INPUTS])
+ downPressed :bool ([NUM_INPUTS])
+ downHeld :bool ([NUM_INPUTS])
+ downReleased :bool ([NUM_INPUTS])
+ upPressed :bool ([NUM_INPUTS])
+ upHeld :bool ([NUM_INPUTS])
+ upReleased :bool ([NUM_INPUTS])
+ rotatePressed :bool ([NUM_INPUTS])
+ rotateHeld :bool ([NUM_INPUTS])
+ antirotatePressed :bool ([NUM_INPUTS])
+ antirotateHeld :bool ([NUM_INPUTS])
+ pausePressed :bool ([NUM_INPUTS])
+ pauseHeld :bool ([NUM_INPUTS])
+ ghostPressed :bool ([NUM_INPUTS])
+ ghostHeld :bool ([NUM_INPUTS])
+ holdPressed :bool ([NUM_INPUTS])
+ holdHeld :bool ([NUM_INPUTS])
+ dropPressed :bool ([NUM_INPUTS])
+ dropHeld :bool ([NUM_INPUTS])
+ screenshotPressed :bool ([NUM_INPUTS])
+ screenshotHeld :bool ([NUM_INPUTS])
util::TextHelper
- xfb :void*
- rmode :GXRModeObj*
+ BLACK :int = 0 {readOnly}
+ RED :int = 1 {readOnly}
+ GREEN :int = 2 {readOnly}
+ YELLOW :int = 3 {readOnly}
+ BLUE :int = 4 {readOnly}
+ MAGENTA :int = 5 {readOnly}
+ CYAN :int = 6 {readOnly}
+ LIGHT_GREY :int = 7 {readOnly}
+ DARK_GREY :int = 8 {readOnly}
+ BRIGHT_RED :int = 9 {readOnly}
+ BRIGHT_GREEN :int = 10 {readOnly}
+ BRIGHT_YELLOW :int = 11 {readOnly}
+ BRIGHT_BLUE :int = 12 {readOnly}
+ BRIGHT_MAGENTA :int = 13 {readOnly}
+ BRIGHT_CYAN :int = 14 {readOnly}
+ WHITE :int = 15 {readOnly}
+ DEFAULT_BACKGROUND :int = BLACK {readOnly}
+ DEFAULT_FOREGROUND :int = WHITE {readOnly}
- TextHelper()
+ init() :bool
+ gotoxy(short, short) :void
+ setColours(int, int) :void
+ useDefaultColours() :void
+ clearScreen() :void

Illustration 8: Platform specific classes

6 Namespaces

Illustration 9 below shows the namespaces into which Toddtris' classes are organised and the relationships between them. Only those classes which are conceptually public (i.e. used by classes in other namespaces) are shown.

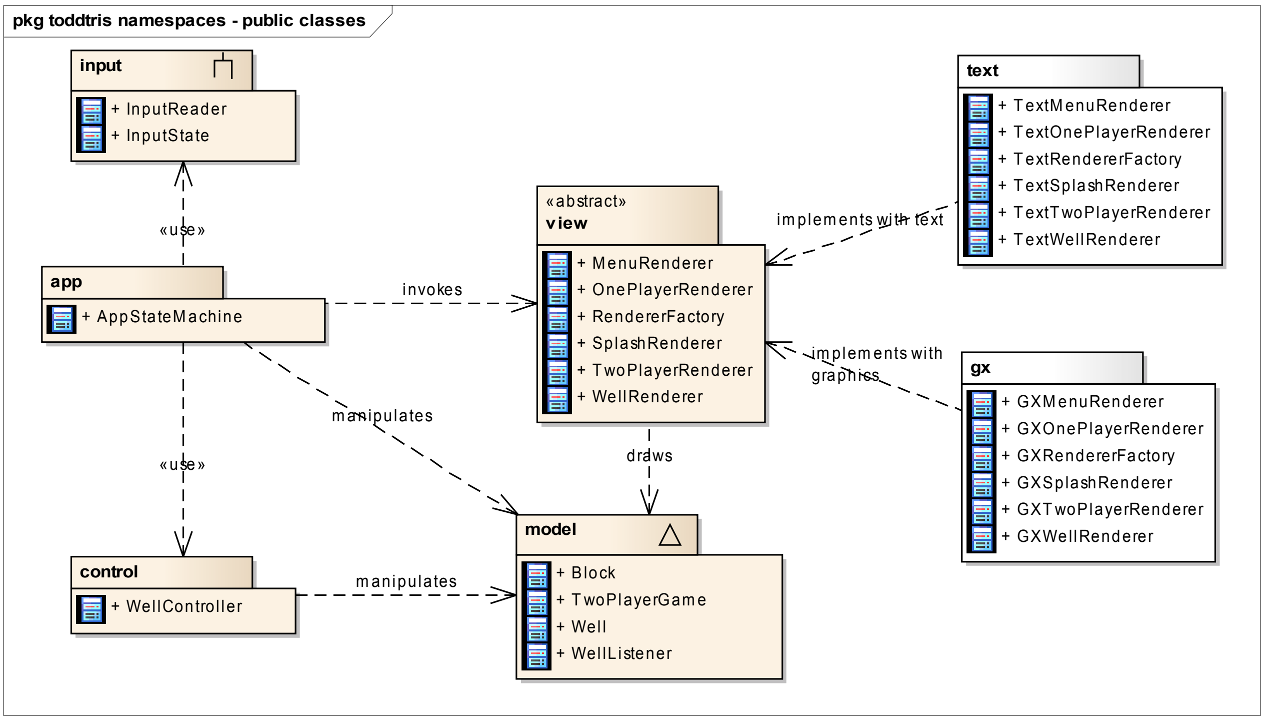

Illustration 9: Toddtris namespaces and 'public' classes

Illustration 10 Shows the the Toddtris namespaces with all classes. The 'util' namespace is also included here; this namespace contains utility classes to find the current system time, implement a timer facility and handle paths to files.

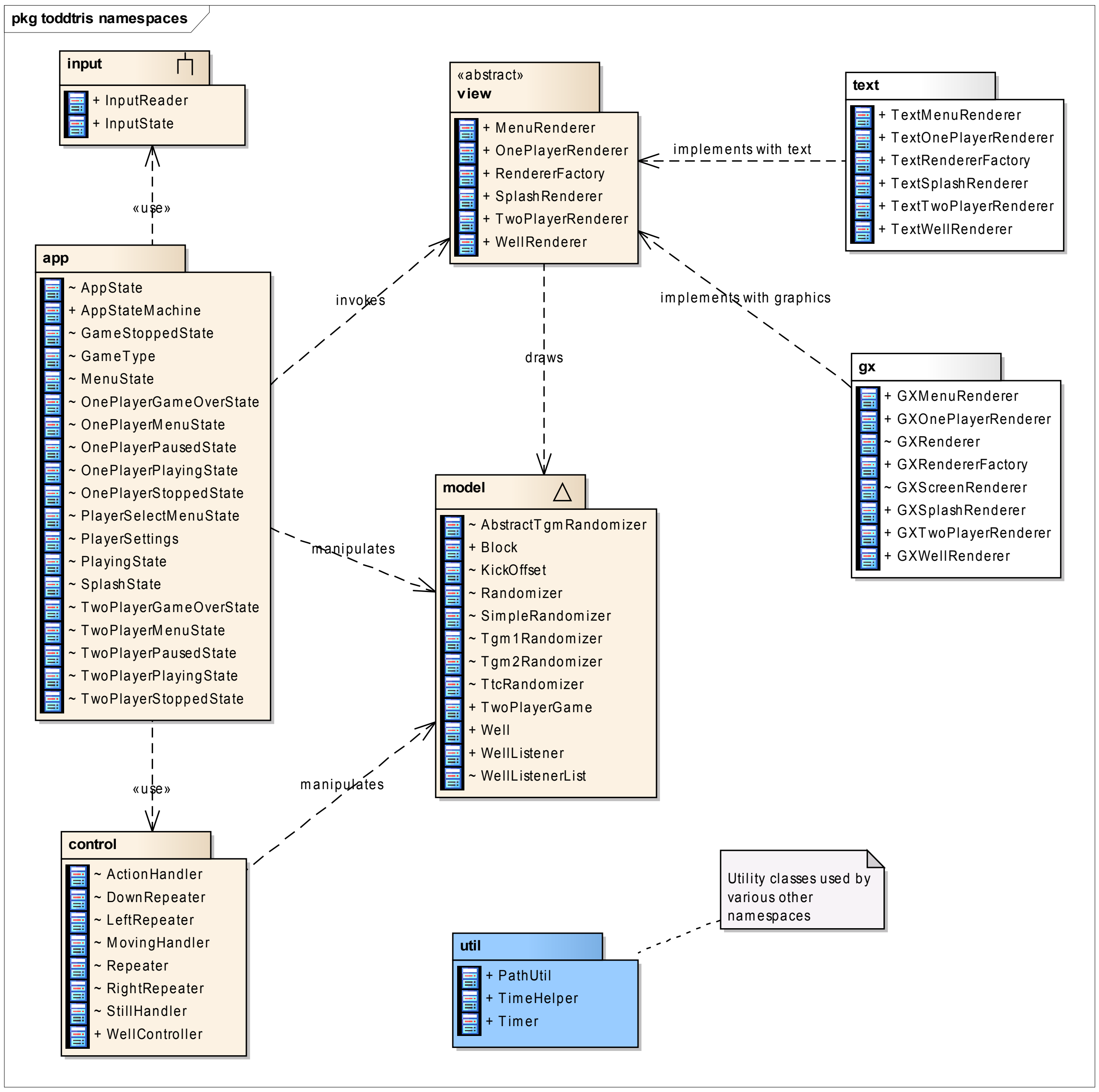

Illustration 10: Toddtris namespaces showing all classes