TheGridGame.cs

This class is the base game class for the Grid Game.
It handles everything from, graphics, game logic, input management, etc.
Only one instance of it is ever running. Contained within are all the different aspects of the game.

- The power model (as its own class)
- Input handling via `MouseState` and `KeyboardState`
- Drawing everything, including calling the draw functions of the objects contained within.
- The entire UI except information on the active bus and batteries and lines.

The game runs in an infinite loop. The two functions `Update(GameTime gameTime)` and `Draw(GameTime gameTime)` get called 60 times a second.
Game Logic and Drawing of things should be kept separate. For example displaying the UI would be handled in Draw(...), but handling player input should be handled in Update(...).

The game itself can be in different `GameState`s, namely:

- Main Menu
- Playing
- Lost
- Won
- LevelSelect
- PauseMenu
- HowToPlay
- Options
- Credits

These tell the game what to render and how to handle game logic. So for example if the game was in `LevelSelect` it shouldn't attempt to solve a power model as it has not been initialized yet.

Class Fields

Name	Type	Explanation
graphics	`GraphicsDeviceManager`	The graphics variable can be used to quickly get information about screen size and other things. Often used in relative positioning problems.

Name	Type	Explanation
spriteBatch	`SpriteBatch`	The SpriteBatch which is used to draw everything. Once the spritebatch is started with `begin(...)` it draws everything issued to it.
theText	`SpriteFont`	The SpriteFont that handles every regular text that appears in the game.
mainMenuFont	`SpriteFont`	This SpriteFont uses the same font as `theText` but is a larger font size. It is used to draw the name of the game in the main menu.
buttonSound	`SoundEffect`	This is the sound effect that plays when you press a button.
buttonSoundInstance	`SoundEffectInstance`	This is an instance of the button press sound effect used in the whole game to not have overlapping sound. Also there is just one instance necessary.
screenCenter	`Vector2`	A Vector that contains the center of the screen. useful if there are buttons that are centered on the middle vertical line.
losingCondition	`bool`	This describes whether or not the player is in a losing state. If it is `true` then the countdown begins.
elapsedTime	`float`	This is used to get the elapsed time in seconds. Handy because you don't have to write `(float)gameTime.ElapsedGameTime.TotalSeconds` each time you access that information.
losingCountdown	`private float`	This variable is used to count down from `losingCountdownFrom`. If it reaches 0 the player loses.
losingCountdownFrom	`float`	Where the countdown starts from. Ranges from 10s (Hard) to 18s (Easy).
countdownStartTime	`float`	The time when the countdown is started. Used as a reference point to see how long is remaining in the countdown. It is set to a new value each time a losing countdown starts.

Name	Type	Explanation
countdownStarted	`bool`	This bool is set to `true` when the system overload countdown has started.
NONE	`const int`	Same constant as in PowerModel.cs, is = `999` and is the value `activeBattery` is assigned to when none is selected.
paused	`bool`	Shows whether the powerModel is paused or not. If `true` the game essentially pauses but does not bring up the Pause Menu.
pauseKeyDown	`bool`	This is `true` when the player presses ‘P’ and will initiate a pause, setting `gameState` to `PauseMenu`.
powerEvent	`bool`	The update function checks for this bool in `powerModel` each frame, if it is true, the game fetches the text of the Event, displays it and pauses the game.
eventTriggered	`bool`	This bool shows whether an Event has been triggered or not. If it has been triggered, the game will not pause for the event again.
eventWasOnBevorMenu	`bool`	Is `true` if an event occured and the event text is shown bevor you changed to menu.
gameState	`GameState`	The state the game is currently in. This is an enum. Can either be: MainMenu, Playing, Lost, Won, LevelSelect, PauseMenu, HowToPlay, Options or Credits.
howToPlayState	`HowToPlayState`	This state is used to tell the game when pressing “Back” in How to Play if it should resume the game or go back to the main menu.
previousClickState	`ButtonState`	This shows if the mouse has been clicked before or not.
lineTextOn	`bool`	If this is set to `true` the lines will display the text. In the Update function the game checks for ‘T’ being pressed, if it is then the PowerModel will not display the text on lines anymore.

Name	Type	Explanation
keyPressed	`bool`	This flag makes sure that keypresses only register once. If a keypress registers this is set to `true` and the condition fails in other checks.
barPosition	`Rectangle`	The rectangle describing the position of the bar in the lower part of the screen.
busInfoPosition	`Rectangle`	This rectangle describes the bounds of the bus information panel.
batteryInfoPosition	`Rectangle`	This rectangle describes the bounds of the battery information panel. It is hidden if there are no batteries or none is selected. It also automatically resizes in case a battery can be changed manually.
menuBackground	`Sprite`	The background picture of the main menu.
selectedRectangle	`Texture2D`	The black rectangle which is displayed around the selected bus.
selectedLevel	`string`	This string is the title of the selected level in the level select menu and is also passed onto powermodel in `powerModel.loadLevel(selectedLevel)` when pressing play.
levelDescription	`string`	This string gets copied from the level files into the level select screen after a level has been selected. It provides a general description of the level.
howToPlay	`string`	This string gets read from here
levelsDirectory	`DirectoryInfo`	The directory in which the levels are saved. This is used to read the levels that are available.
selectedLevelSet	`int`	This int shows which page in the level select menu is currently selected.
maxLevelSets	`int`	This int shows how many pages of levels there are in the level select menu.
levelSelectButtons	`List<UIButton>`	The buttons of the different levels. These get initialized automatically depending on how many levels there are in the Level folder.

Name	Type	Explanation
mainMenuButtons	`List<UIButton>`	The 5 buttons in the main menu.
pauseMenuButtons	`List<UIButton>`	The 5 buttons in the pasue menu.
howToPlayBack	`UIButton`	The “back” button in several menus.
menuButton	`UIButton`	The little button at the bottom right of the screen when playing, that says “MENU”.
playLevelButton	`UIButton`	The Play button in the level select screen.
goBackButton	`UIButton`	Another go back button.
nextLevels	`UIButton`	The `->` button in the level select menu.
previousLevels	`UIButton`	The `<-` button in the level select menu.
batteryButtons	`List<UIButton>`	The buttons that appear when a battery is interactive.
disableSoundButton	`UIButton`	The button that disables sound in the options.
losingCountdownDifficulty	`UIButton`	The button that changes the difficulty in the options.

Class constructor

```
public TheGridGame()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
            graphics.IsFullScreen = false;
            graphics.PreferredBackBufferHeight = 768;
            graphics.PreferredBackBufferWidth = 1024;
        }
```

Input Arguments

None

Description

This is called once when the program starts. It creates a new instance of the Game class, sets it to windowed mode with resolution 1024x768 and sets the root directory of content.

Methods

```
protected override void Initialize()
protected override void LoadContent()
protected override void UnloadContent()
private void beginPause()
private void endPause()
private void checkPauseKey(KeyboardState keyboardState)
public void handleLosingCondition(GameTime gameTime)
protected override void Update(GameTime gameTime)
private void drawRectangle(Rectangle coordinates, Color color)
private void drawBackgroundRectangle(Rectangle coordinates, Color color)
private void drawDeeperBackgroundRectangle(Rectangle coordinates, Color color)
private void drawPauseMenu()
private void drawMainMenu()
private void drawBatteryButtons()
private void drawEndMessage(GameState endState)
private void drawLevelSelectMenu()
private void drawHowToPlay()
private void drawOptions()
private void drawCredits()
protected override void Draw(GameTime gameTime)
```

Initialize

```
protected override void Initialize()
```

Input Arguments

None

Description

This method is called once when an instance of the game class is created.
It sets the gameState to MainMenu, initializes all variables to a state which make sense for a fresh game.
The DirectoryInfo `levelsDirectory` gets set to the proper directory where the levels are saved.
This is later used to read the files inside the folder "Level". This means the player can create their own level by adding a new textfile called Level_*.txt to the folder.
This textfile should be written according to the template provided, otherwise it will not work.
The lists of buttons are initialized and then level select buttons is populated according to the graphics shown in the description of `drawLevelSelect()`.

As soon as six buttons are created, the Y position will reset and another group of six will be placed on top of eachother.
Each group of six buttons then represents one level set.
The other buttons are all statically placed, their placement can be seen in the Draw methods of the respective screens.

LoadContent

```
protected override void LoadContent()
```

Input Arguments

None

Description

This method loads game content that is necessary for the game to display things before any proper gameplay has started.
This includes:

- Creating a new `SpriteBatch`.
- Loading the SpriteFonts for game text and main menu title text.
- The background picture for the menu.
- The black rectangle placed around buses when they are selected.
- The sound effect and its instance when a button is clicked.

All other Sprites are loaded whenever a level is loaded to avoid unnecessary overhead.

UnloadContent

```
protected override void UnloadContent()
```

Input Arguments

None

Description

UnloadContent is currently not in use but is placed there as a placeholder in case memory gets too full at some point, this can then be called to remove unnecessary Sprites from memory to make space for others.
However, currently the amount of images loaded remains small enough for it to not become a problem.

---

beginPause

```
private void beginPause()
```

Input Arguments

None

Description

This method sets the `paused` flag to true, which pauses the powerModel. Then it sets `gameState` to `PauseMenu` which brings up the menu and starts checking for user input on the menu buttons.

If an event occured before this method started, it sets `eventWasOnBeforeMenu` to `true` for saving that state and `powerEvent` to false so that the event text is hidden.

---

endPause

```
private void endPause()
```

Input Arguments

None

Description

This essentialy reverts the state changes made by `beginPause()` and resumes the powerModel by setting `paused = true`.
If an event occured before the pause, the event will be shown again by setting `powerEvent` to `true`. In this case it also resets `eventWasOnBeforeMenu` and pauses the game again for the event.

checkPauseKey

```
private void checkPauseKey(KeyboardState keyboardState)
```

Input Arguments

- `keyboardState`: This is the current state of the keyboard.

Description

This method gets called in `Update(gameTime)` once per frame. It checks if the 'Escape' key is pressed and if it is it will initiate/stop the pause.
At the end of the method `pauseKeyDown` is set to `pauseKeyDownNow` so that in case the user has not managed to release the pause key in one frame (which is likely to happen) the game will not try to unpause again.
Otherwise the game would toggle pause every 1/60th of a second.

---

handleLosingCondition

```
public void handleLosingCondition(GameTime gameTime)
```

Input Arguments

- `gameTime`: The current time in the game.

Description

This method manages the countdown in case the PowerModel returns `true` for `losingCondition()`.
First it checks that `losingCondition` is `true` and that the countdown has not started yet. If both apply it will set the flag that the countdown has started, initializes `countdownStartTime` to the current time in seconds and sets an initial value of `0f`
to `losingCountdown`.
Then each time `Update(gameTime)` calls this function, the countdown is updated by subtracting the elapsed time in seconds from the difficulty setting.
If the countdown reaches 0, `gameState` is set to `Lost` and the player has to select a new level.

In case `losingCondition` is `false` again (which means the player has managed to balance the system), the countdown is reset and hidden again.

Update

```
protected override void Update(GameTime gameTime)
```

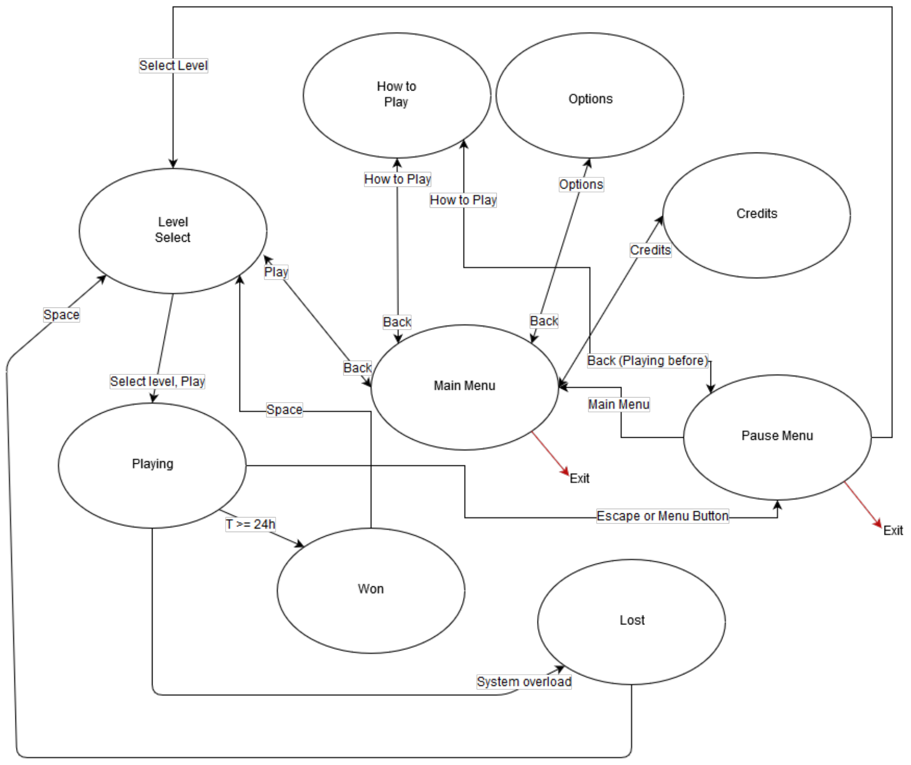
Input Arguments

- `gameTime`: The current time in the game.

Description

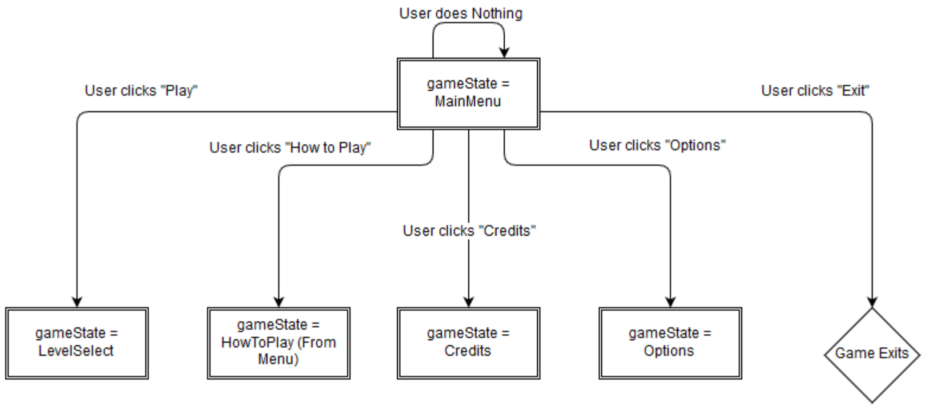
This function handles all game logic except graphics. For example it checks for button clicks, user inputs, etc.
It is automatically called 60 time per second.

First it creates an instance of `MouseState` and `KeyBoard` state, which is essentially a "snapshot" of the input devices.
Then it treats user input according to what the current `gameState` is.
The transition between the different `GameState` is shown in this diagram:

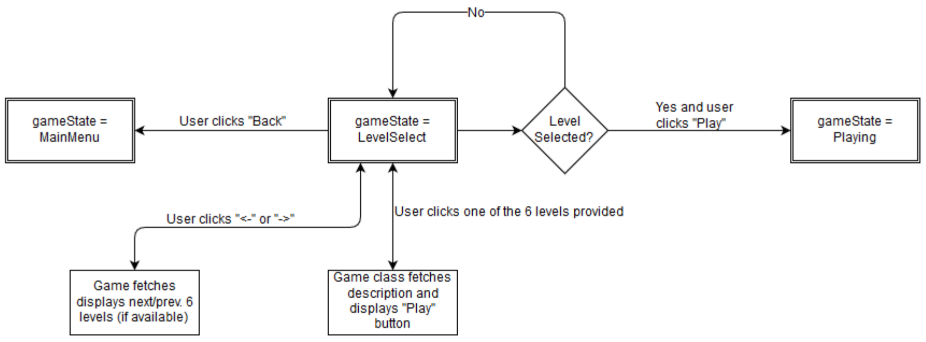


The following will show how the different game states handle input.

Main Menu
The main menu consists of 5 buttons.
Each buttons `Update(MouseState mouseState` function is called. This is responsible for changing the color of the button slightly if the mouse hovers over it.
Then if the left mouse button is clicked, `keyPressed` is set to true to avoid multiple frames processing the click.
Each button contains a `Rectangle` which is provided by the Framework, this checks if the position of the mouse is within its boundaries during the click.
If it is, it triggers the corresponding `if` condition.
The following state diagram shows how the states transition:



Level Selection Menu
As in the main menu, each available button is Updated to show color change on mouseover.
Then if there are more than 6 levels, the `->` and `<-` buttons work.
Each group of 6 levels is put into a level Set. And only the buttons for the current level set are updated.
For example if level set is 2, the for loop skips 2*6 = 12 buttons before allowing user interaction.
The `Play` button does not appear until a level has been selected. Clicking it will call `loadLevel(...)` of the power model and set the state to playing
The state transition is as shown on this diagram:


gameState = MainMenu
User clicks "Back"
gameState = LevelSelect
No
Level Selected?
Yes and user clicks "Play"
gameState = Playing
User clicks "<-" or "->"
User clicks one of the 6 levels provided
Game fetches displays next/prev. 6 levels (if available)
Game class fetches description and displays "Play" button

Options Menu

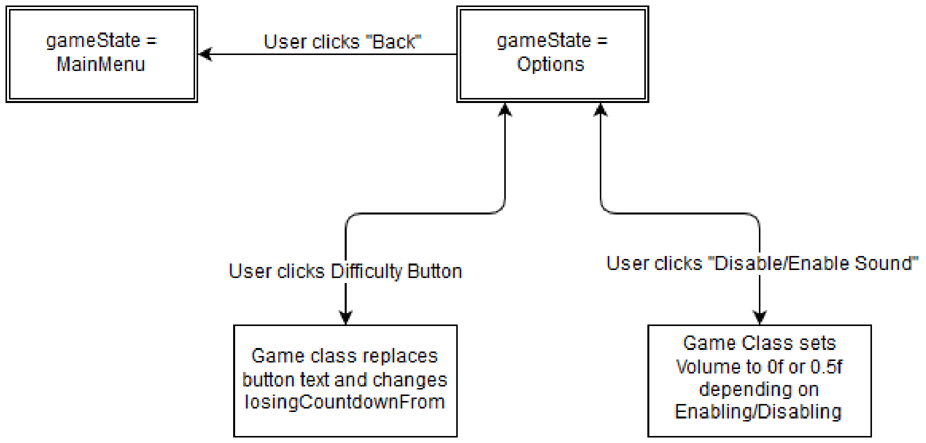
The options menu presents the player with two options:

- Disable Sound
- Change Difficulty

Changing the difficulty will increase the countdown time until the system breaks. The buttons are updated to show the mouse hovering over the button.

If the player clicks the difficulty button, the text on the button is replaced with the next difficulty and `losingCountdownFrom` is changed accordingly.

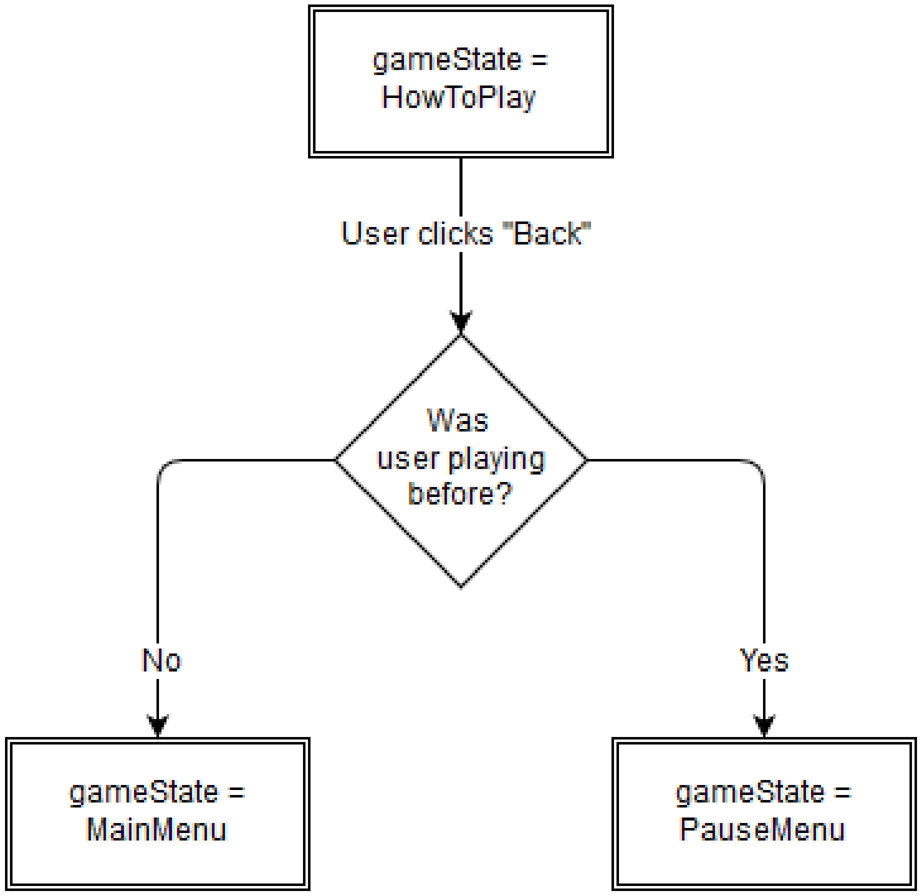
The state transition diagram is as follows:


gameState = MainMenu
User clicks "Back"
gameState = Options
User clicks Difficulty Button
User clicks "Disable/Enable Sound"
Game class replaces button text and changes losingCountdownFrom
Game Class sets Volume to 0f or 0.5f depending on Enabling/Disabling

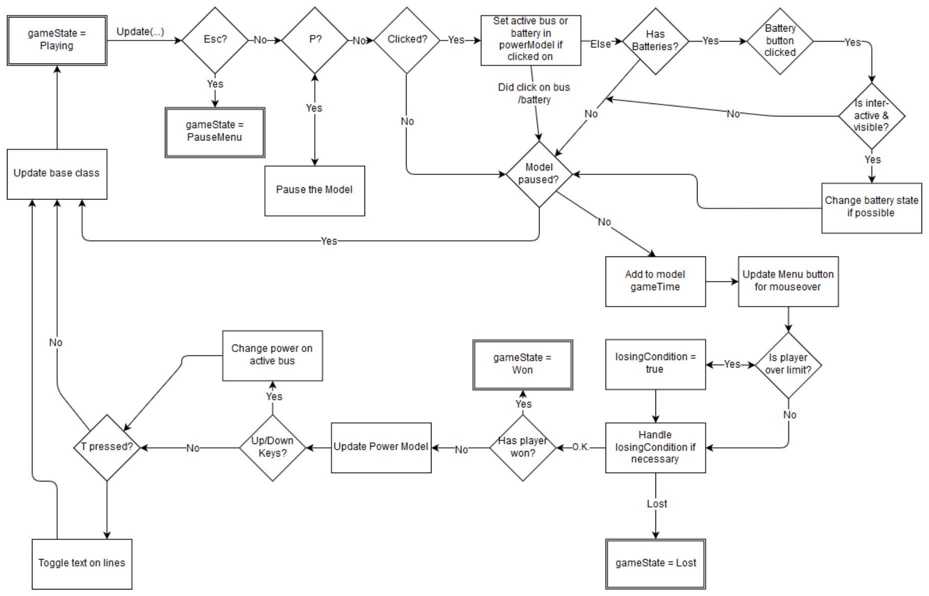
How to Play

The how to Play screen displays the text describing how to play the game.
The only the the user can do here is press `Back` to go back to the previous screen.
It is important to check if the user was playing a level before or came here from the main menu.
If the player came from the main menu he should be brought back to the main menu, as the game class cannot resume a power model that isn't initialized.
`gameState = (howToPlayState == HowToPlayState.FromMenu) ? GameState.MainMenu`

`: GameState.PauseMenu;` is a shortened if/else condition, which assigns `gameState = GameState.MainMenu` if HowtoPlayState is `FromMenu` or `gameState = GameState.Playing` if the user was playing before.



Playing
While the user is playing a plethora of things are checked. It is explained in the diagram below.
The base game Update needs to be called in any case because it keeps track of time and other necessary things.



Won

The player simply has to press space to be sent back to the level selection menu.

Lost

The player simply has to press space to be sent back to the level selection menu.

---

drawRectangle

```
private void drawRectangle(Rectangle coordinates, Color color)
```

Input Arguments

- `coordinates`: A rectangle containing position (of the upper left corner) and dimensions of the rectangle to be drawn.
- `color`: The color of the rectangle to be drawn.

Description

This method draws a rectangle over the complete area of the prodived rectangle `coordinates`.
It is on layer `0.2f` so in the layer closest to the player of the three available rectangles.

---

drawBackgroundRectangle

```
private void drawBackgroundRectangle(Rectangle coordinates, Color color)
```

Input Arguments

- `coordinates`: A rectangle containing position (of the upper left corner) and dimensions of the rectangle to be drawn.
- `color`: The color of the rectangle to be drawn.

Description

This method draws a rectangle over the complete area of the prodived rectangle `coordinates`.
It is on layer `0.25f` so in the layer inbetween `drawRectangle(...)` and `drawDeeperBackgroundRectangle(...)`.

---

drawDeeperBackgroundRectangle

```
private void drawDeeperBackgroundRectangle(Rectangle coordinates, Color color)
```

Input Arguments

- `coordinates`: A rectangle containing position (of the upper left corner) and dimensions of the rectangle to be drawn.
- `color`: The color of the rectangle to be drawn.

Description

This method draws a rectangle over the complete area of the prodived rectangle `coordinates`.
It is on layer `0.26f` so in the rectangle layer furthest away from the player.

---

drawPauseMenu

```
private void drawPauseMenu()
```

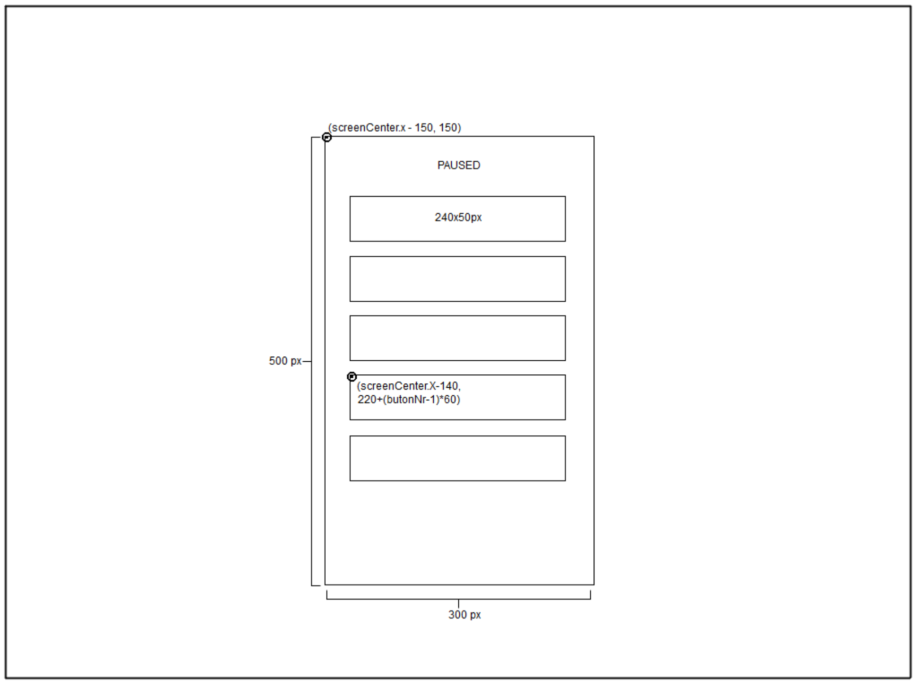
Input Arguments

None

Description

This method draws all graphical elements contained in the pause menu.
This function is overlaid on the Graphics of playing, so the menu appears "on top" of the gameplay.
All text is centered on the buttons or the rectangle on which the text is drawn. This is done using the method `MeasureString(string)` of `SpriteFont`, which returns a `Vector2` containing size information about the text width and height.

As a guideline for how the positioning of the elements looks like, please refer to this guide:



---

drawMainMenu

```
private void drawMainMenu()
```

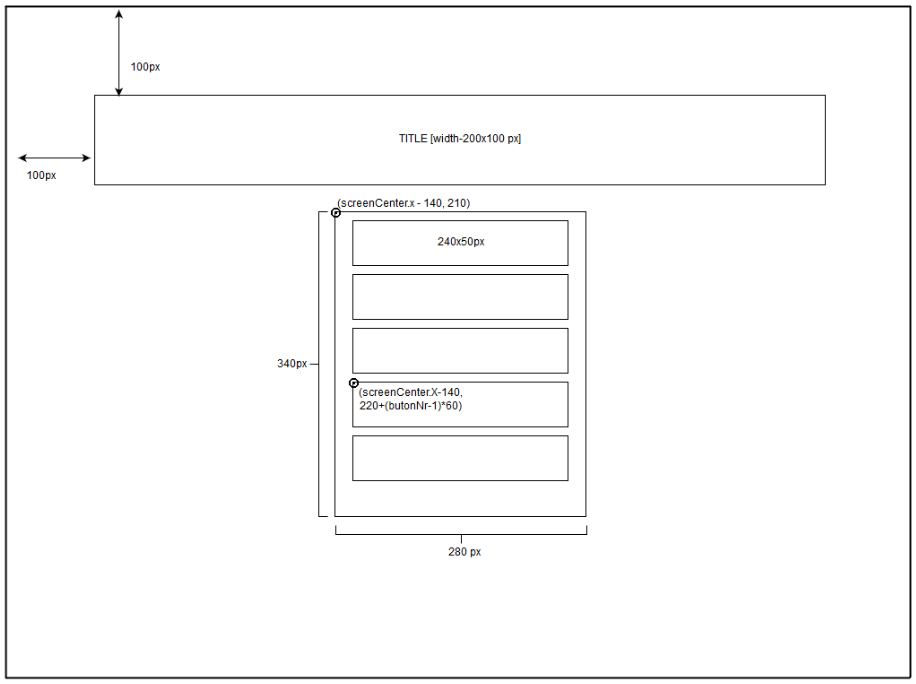
Input Arguments

None

Description

This Method is responsible for drawing the main menu.
All text is centered on the buttons or the rectangle on which the text is drawn. This is done using the method `MeasureString(string)` of `SpriteFont`, which returns a `Vector2` containing size information about the text width and height.

As a guideline for how the positioning of the elements looks like, please refer to this guide:


100px
TITLE [width-200x100 px]
100px
(screenCenter.x - 140, 210)
240x50px
340px
(screenCenter.X-140, 220+(butonNr-1)*60)
280 px

---

drawBatteryButtons

```
private void drawBatteryButtons()
```

Input Arguments

None

Description

This method draws the buttons on the active battery panel, in case the battery is interactive.
It will only be called if the battery selected is considered an active battery.

The positioning of the buttons is done by hand and the text scaling too, that's why absolute numbers are used.

---

drawEndMessage

```
private void drawEndMessage(GameState endState)
```

Input Arguments

- `endState`: This is the gameState in the instance of winning/losing. Depending on what this is, a winning or losing message is drawn.

Description

This method draws the winning/losing screen.
Depending on whether `gameState` is `Won` or `Lost`, it will inform the player of them winning or losing.
The rectangle is centered on the screen and its size is calculated based on the size of the text.

---

drawLevelSelectMenu

```
private void drawLevelSelectMenu()
```

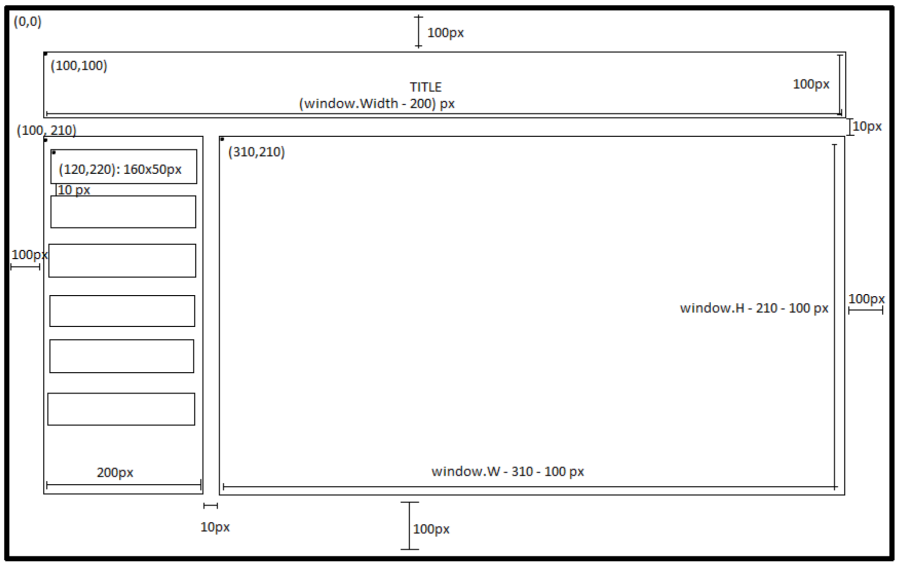
Input Arguments

None

Description

This method draws the level selection screen.
It draws rectangles for the necessary panels and then only draws the buttons for the levels which are in the current level set.
Above the `->` and `<-` button the current level set selection is shown.
Once the player selects a level, its `levelDescription` is read in the `Update(...)` function and displayed on the panel on the right, along with its title.
The level buttons are grouped into pairs of six, so depening on which set of levels is selected, it will skip the irrelevant ones.
The `Play` button is only drawn once the player has selected a level, as it is not interactive in `Update(...)` until the player has selected a level.
This is done to avoid potential confusion.

To see the placement of elements, please refer to the following picture:



---

drawHowToPlay

```
private void drawHowToPlay()
```

Input Arguments

None

Description

This method draws the how to play screen.
First it draws a rectangle from Position (100,100) with the dimensions (screenWidth - 200 x screenHeight - 200), then it draws the string `howToPlay` which contains the instructions for the game.
The text can be found in the folder "Manual" in "Content".
Finally it draws the "Back" button with the text on it.

---

drawOptions

```
private void drawOptions()
```

Input Arguments

None

Description

This method draws the options screen.
First it draws a rectangle from Position (100,100) with the dimensions (screenWidth - 200 x screenHeight - 200), then it draws the “Back” button. Note that this is the same button as in “How to Play”, as the position is the same and they both serve the same function.
After that it measures the string on the button `howToPlayBack` and centers it on the button.
Finally it draws the option buttons and a string describing what the difficulty button does.

---

drawCredits

```
private void drawCredits()
```

Input Arguments

None

Description

This method draws the options screen.
First it draws a rectangle from Position (100,100) with the dimensions (screenWidth - 200 x screenHeight - 200), then it draws the “Back” button. Note that this is the same button as in “How to Play”, as the position is the same and they both serve the same function.
After that it measures the string on the button `howToPlayBack` and centers it on the button.
Finally it draws the string containing the game credits.
The for loop splits up the text by the newline character and centers the text on the middle of the screen.

Draw

```
protected override void Draw(GameTime gameTime)
```

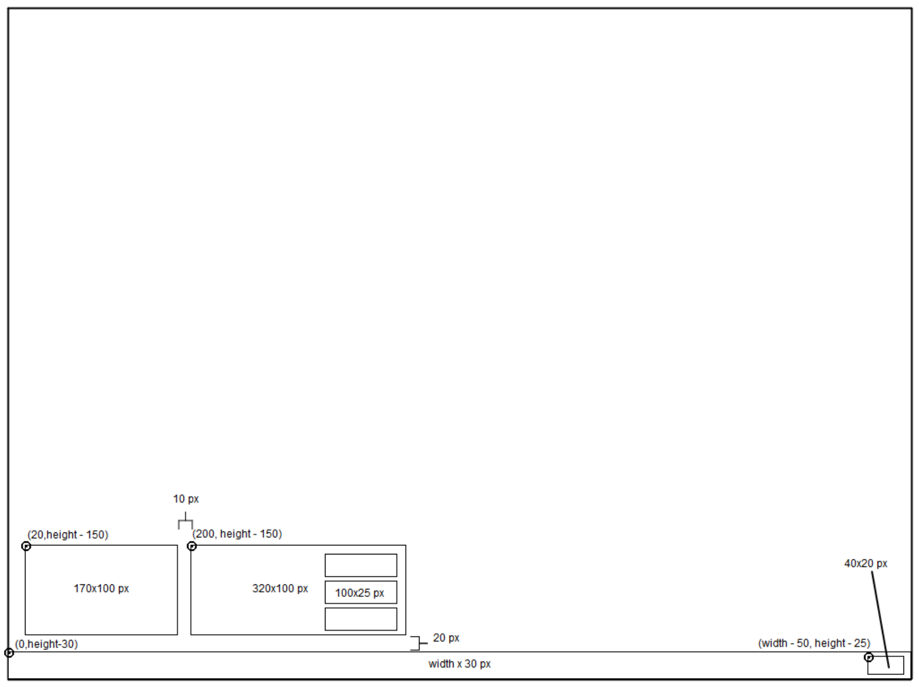
Input Arguments

- `gameTime`: The current time since the program has started.

Description

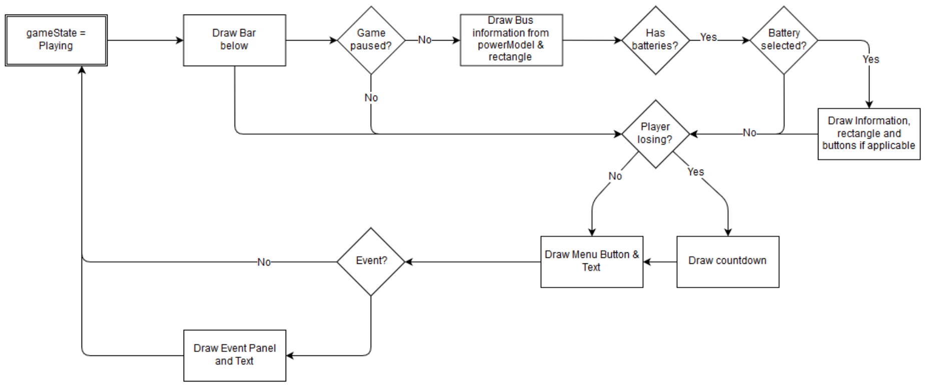
This method calls all necessary drawing functions based on `gameState`.
All states except `Playing` and `PauseMenu` are completely separate screens, so their drawing functions should only be called whenever necessary.
As the pause menu overlaps the gameplay, the last `if` statement contains both cases.

It draws the gameplay interface with the following positioning:



The pause menu is only drawn if the player pauses the game, the positioning of it can be found in the description of `drawPauseMenu()`.

The following diagram shows the decision flow in the Draw function when `gameState == Playing`:


gameState = Playing
Draw Bar below
Game paused?
No
Draw Bus information from powerModel & rectangle
Has batteries?
Yes
Battery selected?
Yes
Draw Information, rectangle and buttons if applicable
No
No
Player losing?
No
Yes
Draw Menu Button & Text
Draw countdown
Event?
No
Draw Event Panel and Text