PowerModel.cs

This class contains everything that is related to the power model used for the transmission lines.
The model uses the DC Power Flow approximation to calculate Line flows within the model.
It also handles some of the user interaction with the power model, such as changing a buses power or
changing the state of batteries.

Class Fields

Name	Type	Explanation
buses	`List<Bus>`	This is the list containing all buses currently in the model.
lines	`List<Lines>`	This list contains all the power lines used in the model.
batteries	`List<Battery>`	This list contains all batteries currently placed in the power model.
removedLines	`List<Line>`	This is the list containing all power lines currently removed from the model. It is there to simplify re-inserting a line into the model again. After the line has been added again, it is deleted from this list.
allEvents	`List<Events>`	This is a list that contains all the events for the currently active level.
currentID	`int`	This integer saves which bus ID is to be assigned next when adding a new bus. Each time a bus is added it is increased by 1.
currentLineID	`int`	This is the same kind of integer as currentID, but for lines.

Name	Type	Explanation
gDevice	`GraphicsDevice`	The current graphics device from the main game class. This is used because making a new sprite needs the graphics device to load a new texture. By saving this it isn't necessary anymore to pass the graphics device into the power model each time a bus is added, thus uncoupling the two classes for changing the model.
gManager	`ContentManager`	This is a copy of the content manager from the base class. This is also saved in the model to make loading textures easier.
bSolved	`Matrix<double>`	The solved B-Matrix of the DC Power Flow approximation
modelSolved	`bool`	A boolean which is set to `true` in case the power model has been solved and no changes have been made. This is done to reduce unnecessary solving.
currentLines	`int`	This keeps track of how many lines there are in the model.
solvedLines	`int`	This keeps track of how many lines have been solved for in the model. In case a new one is added the B Matrix gets recalculated.
idEvent	`int`	ID of the event which is runnig. This is used to get the right event out of allEvents.
timeInLevel	`float`	The current time in the level in minutes.
timeInLevelHour	`float`	The current time in hours.
startTime	`float`	This saves the time when a new level is started. This is done so the level time can be used relative to the starting time.
showEventText	`bool`	This bool is set to true whenever an Event occurs. The main class checks for it being true in the method `eventOccurs()`. If it is the event text is displayed on the screen.

Name	Type	Explanation
activeBus	`int`	The ID of the bus which the player has currently selected.
activeBattery	`int`	The ID of the battery which is currently selected.
NONE	`const int`	A constant with the value 999. Active battery is set to this if no battery is selected. This is so that there is no “magic number” representing no active battery.
mouseClicked	`bool`	A boolean that is set to true if the mouse has been clicked. This is done so that mouse clicks only register for one frame and don’t get processed for any following frames.
windEventStartingTime	`float`	Game time at which a wind event starts.
windEventEndingTime	`float`	Game time at which the started wind event ends.
windScalingFactor	`float`	Factor of the maximal power (P_min) the wind turbine is going to generate in the middle of the event.
solarScalingFactor	`float`	Factor of the maximal power (P_min) the photovoltaic system is going to generate in the middle of the event.
solarEventEndingTime	`float`	Time at which the photovoltaic generation is back at its normal course.
arrow	`Texture`	The texture of the arrow used on the power lines. This is saved in the power model class to provide dependency injection for the arrow on the line, otherwise a new arrow texture would have to be generated for each individual line.
gameBackground	`Sprite`	The background texture of the game. This is created with position (0,0) (upper left corner) and is drawn in the background plane. It can be set in the level file.

Name	Type	Explanation
BatteryChargeState	`enum`	This enum is used to show the charging state of an individual battery. The three states are `Charging`, `Discharging` and `Idle`.
ActiveBattery	`int`	This is the public property of `activeBattery` so no “get” method has to be written for it.

Class constructor

```
public PowerModel(GraphicsDevice gDevice, ContentManager cManager)
```

Input arguments

- `gDevice`: The current graphic settings of the game class. In case the graphics change in the meantime a new instance is needed.
- `cManager`: The content manager of the game. This is used to load new assets into objects or is passed into objects to load assets in them.

Description

The constructor initializes all fields of the PowerModel class that are dynamic, such as buses and lines etc..
It also sets `currentID` and `currentLineID` to zero.
It loads the arrow image into the `arrow` texture, initializes `background` and loads the default background image into it.
Any other variable keeping track of the current power model is set to the values one would expect of a “fresh” model.

Methods

```
public void loadLevel(string level, float elapsedTime)
public void solve()
private double[] getPowersArray()
private double[,] getBMatrixArray()
public void addEvent(int eventID, int busID, float start, float end,
    double P, string text)
public void loadEvent()
public bool eventOccurs()
public string getEventText()
public void clearEventText()
public void addBus(Vector2 position, double P, double PMax, double
    PMin, double U, double Theta, string Type)
public void addLine(int start, int end, double pMax, double R, double X)
public void addBattery(int busID, double power, double eMax, double
    initialCharge, double chargeCapacity, char allowsChange)
public void removeLine(int id)
public void restoreLine(int id)
public void setBusPower(int busID, double power)
public void eventChangeBusPower(double amount)
public void dynamicWindChange()
public void dynamicPhotovoltaicChange()
public void updateBatteries()
public void modifyBatteryState(int batteryID, String newState)
public bool isActiveBatteryChangeable()
public void manualChangeBusPower(double amount)
public void toggleLineText()
public void increaseLinePower()
public bool containsBatteries()
public Boolean losingCondition()
public Boolean winCondition()
public void Update(MouseState mouseState, float elapsedTime)
public void checkMouse(MouseState mouseState)
public Vector2 getActiveBatteryPosition()
public void drawBatteryInfo(SpriteBatch spriteBatch, SpriteFont theText)
public Vector2 getActiveBusPosition()
public void drawBusInfo(SpriteBatch spriteBatch, SpriteFont theText)
public void Draw(SpriteBatch spriteBatch, SpriteFont theText)
public String toString()
```

---

loadLevel

```
public void loadLevel(string level, float elapsedTime)
```

Input Arguments

- `level`: The level which is loaded. Its format should be “Level_x” whereas x is the level number.
- `elapsedTime`: The current elapsed time in seconds of the whole program. It starts as soon as the program starts.

Description

First `loadLevel` clears the old level. Then it sets the `startTime` of the level. Afterwards, it loads the content of the new level from the text file in Content\Level.
With these loaded parameters `loadLevel` calls `addBus`, `addBattery`, `addLind`, `addEvent` and `gameBackground.LoadContent`.

---

solve

```
public void solve()
```

Input Arguments

None

Description

This method solves the power flows of the current model using the DC power flow algorithm.
The method exits if the model has already been solved or if the current amount of lines is 0.
Next it builds the vector P containing all Powers beside the slack bus. Using the function `getPowersArray()` it converts the Array into the Vector.

In case another line has been added the following happens:
The 2-dimensional Array `bMatrixArray` is then built using `getBMatrixArray()`, converted into a `Matrix` and saved.
`solvedLines` is set to `currentLines` and unless `currentLines` changes, these steps are skipped in the next iteration.

Next the Theta Vector is built using a simple for-loop.

Then the Thetas of the buses are solved using Theta = B^(-1) * P and assigned to the proper buses.

The line power flows are solved using equations `ADD EQUATIONS`.

Finally the power of the slackbus is adjusted to compensate for all changes in the other buses.
`modelSolved` is set to true to avoid re-solving an already solved model.

---

getPowersArray

```
private double[] getPowersArray()
```

Input Arguments

None

Returns

An array containing the powers of all buses except the slack bus.

Description

An array is declared of size `buses.Count - 1` and initialized with the powers of all buses besides the slackBus (ID = `1`).

---

getBMatrixArray

```
private double[,] getBMatrixArray()
```

Input Arguments

None

Returns

A 2-dimensional double array which represents the B matrix.

Description

First a 2-dimensional array of size `buses.Count - 1 X buses.Count - 1` is declared.
It is then filled as per the DC Power Flow algorithm. Diagonal Elements B__ii are the sum of all inverse inductive reactances connected to that bus.
All other B__ij are the negative inverse reactances of said line connecting bus i & bus j. If no such line exists, B__ij = 0.

---

addEvent

```
public void addEvent(int eventID, int busID, float start, float end,
double P, string text)
```

Input Arguments

- `eventID`: The ID of the event.
- `busID`: The ID of the bus which should be changed.
- `start`: The start time of the event.
- `end`: The end time of the event.
- `P`: The amount of power by which the buses are changed by this event.
- `text`: The text which is shown when the event occurs.

Description

`addEvent` creates a new event. This event will be called later by `loadEvent`. It also adds the event to the list `allEvents`.

---

loadEvent

```
public void loadEvent()
```

Input Arguments

None.

Description

When an events starting time equals the game time, an event is going to occur.

- If its `eventID` is 1 then the bus chosen by `busID` changes its power by the chosen `P`.
- If its `eventID` is 2 then the wind turbines start working at the start time `windEventStartingTime` and end at the end time `windEventEndingTime`. This is done by the method `dynamicWindChange`. The maximal power of the event is thereby `PMax` * `windScalingFactor` which is taken from the `P` of the event.
- If its `eventID` is 3 then the `solarScalingFactor` is set to `P` and the curve constructed by `dynamicPhotovoltaicChange` will change.
- If its `eventID` is 4 then all generators change their power by the amount `P`.
- If its `eventID` is 5 then all consumers change their power by the amount `P`.
- If its `eventID` is 6 then the line with the line ID chosen by `BusID` will be removed by `removeLine(int id)`.
- If its `eventID` is 7 then the line with the line ID chosen by `BusID` will be restored by `restoreLind(int id)`.
- If its `eventID` is 8 then all hydro power plants change their power by the amount `P`.

Then it also sets `showEventText` to `true`, so that the text is shown.

---

eventOccurs

```
public bool eventOccurs()
```

Input Arguments

None

Returns

`true` if there is an Event, `false` otherwise.

Description

This returns true if an event is triggered by the power model. This function is used in the game class `Update(GameTime gameTime)` function.

---

getEventText

```
public string getEventText()
```

Input Arguments

None

Returns

The text description of the current event.

Description

This function is used by the game class to fetch the text description of the current event.

---

clearEventText

```
public void clearEventText()
```

Input Arguments

None

Description

This function sets `showEventText` to `false`. This then ensures that the game class will not display an event text anymore.

---

addBus

```
public void addBus(Vector2 position, double P, double PMax, double
PMin, double U, double Theta, string Type)
```

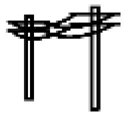
Input Arguments

- `position`: The position on the screen where the bus will be placed.
- `P`: The initial consumption/generation at the bus.
- `PMax`: The maximum consumption/minimum generation of the bus.
- `PMin`: The minimum consumption/maximum generation of the bus.
- `U`: The initial voltage at the bus.
- `Theta`: The initial angle theta at the bus.
- `Type`: The type of the bus. This can be a string of: S, P, W, G, C, H.

Description

The function creates a new instance of `Bus` at `position` using the parameters `P`, `PMax`, `PMin`, `U`, `Theta` and `Type`.
It will also assign an incremental BusID starting from BusID `1`. The current instance of `gDevice` is also passed to the constructor.

Then, depending on the Type given to the method, it will load one of the following images:

- "S": 

- “P”:
- “W”:
- “G”:
- “C”:
- “H”:

Please note that the wind turbine has a separate picture which is loaded separately in the function.
This is because the blades are animated and are spinning. Their speed depends on the current power generation of the bus.

After having loaded the picture of the bus, it also sets the rectangle used for checking whether the mouse is on the bus relative to its position.
Then the image for the status bar above the bus is loaded and saved accordingly.
Finally the bus is added to `List<Bus> buses` and the model is set to not solved.

---

addLine

```
public void addLine(int start, int end, double pMax, double R, double X)
```

Input Arguments

- `start`: This is the ID of the bus where the line starts from.
- `end`: This is the ID of the bus where the line ends.
- `pMax`: The Maximum amount of P allowed to flow over the line.
- `R`: The Resistance of the line in p.u. This is currently not used.
- `X`: The Reactance of the line in p.u.

Description

This method adds a new line which connects two buses to the model.
First it checks for the line being invalid, if either the start or end ID of the line exceeds the current range of IDs, nothing happens.
Then the `Vector2` describing the endpoints of the line are initialized.
Then a new instance of `Line` is created using `currentLineID + 1`, both corresponding busIDs, the positions and `pMax`, `R` & `X`.
Since the arrow texture is not saved in the constructor of `Line` it is injected by using `line.Arrow = arrow`.
After this, the texture of the line is loaded using the 1-pixel texture Sprites/line.
The line is then added to `List<Line> lines` and `currentLines` & `currentLineID` are incremented by 1 to accomodate for the next possible line and the model is set to not solved.

The unique ID `currentLineID` for each line is important because when removing a powerline from the model, it is important to know if the line to be restored is the same that was removed before.

---

addBattery

```
public void addBattery(int busID, double power, double eMax, double
initialCharge, double chargeCapacity, char allowsChange)
```

Input Arguments

- `busID`: The ID of the bus where the battery is attached.
- `power`: The power the battery can deliver when discharging or consumes when charging.
- `eMax`: The maximum energy that can be stored in the battery.
- `initialCharge`: The initial charge the battery holds when a level is started.
- `chargeCapacity`: This defines how quickly the battery charges. A charge capacity of `1` means the battery charges 1 unit of energy in 1 hour.
- `allowsChange`: This char defines whether the battery is interactive for the player or not. Hydro and Generators are always interactive. ‘Y’ and ‘N’ allowed. Any other char will set it to false.

Description

The function exits in case the associated busID is bigger than the amount of buses currently in the system.
Then a copy of the bus associated to the ID is made and initialized accordingly.

The boolean `changeAllowed` is created and set to `true` in case `allowsChange` is 'Y'.
The position of the battery is intially set to the position of the bus, but inside the `Battery` class it is moved by a certain factor so they don't overlap.
A new battery is created using

```
Battery battery = new Battery(busID, power, eMax, initialCharge,
chargeCapacity, batteryPosition, gDevice, changeAllowed);
```

A new 1x1 pixel `Texture2D` is created to represent the line connecting to the bus.
This is a little hack so no new texture has to be loaded. It could also be loaded using a new `Texture2D` and loading `Sprites/line` into it.
This texture is then loaded into the battery object.

Similarily to the bus, the rectangle for interaction with the mouse is set and the image for the status bar is loaded.
The fished battery is then added to `List<Battery> batteries`.

---

removeLine

```
public void removeLine(int id)
```

Input Arguments

- `id`: The unique ID of the line to be removed.

Description

This method exits if the ID of the line to be removed is bigger than the amount of lines currently in the level.
It then iterates over the `List<Line> lines` and checks if the current line ID is equal to the one to be removed the variable `x` keeps track of the current index in the list.
If it is, a copy of the line is made and added to `List<Line> removedLines` and removed at the position `x` in the List of active lines.
The amount of lines currently in the game (`currentLines`) is reduced by one and the model has to be solved again.

---

restoreLine

```
public void restoreLine(int id)
```

Input Arguments

- `id`: The ID of the line to be restored.

Description

Contrary to `removeLine(id)` this method reinstates the line with ID `id` to the model.
If the list of removed lines is empty nothing will happen.
It iterates through the list of removed lines while keeping track of the current index `x`.
If a line matches the ID provided, it gets added to `lines` and gets removed from `removedLines`.
`currentLines` is incremented by one and the model has to be solved again.

---

setBusPower

```
public void setBusPower(int busID, double power)
```

Input Arguments

- `busID`: The ID of the bus to be changed.
- `power`: The new amount of power at the bus.

Description

The `bus.P` is set to `power`.
The powerReference for the battery is set to the previous power of the bus.

---

eventChangeBusPower

```
public void eventChangeBusPower(double amount)
```

Input Arguments

- `amount`: The relative amount of power. Can be positive or negative.

Description

This method changes the power at a bus which is set active for an event. The change is relative to the amount provided.
If the amount is negative, it will be subtracted from the power and vice versa.

---

dynamicWindChange

```
public void dynamicWindChange()
```

Input Arguments

None.

Description

This function simulates a wind event if the `windEventStartingTime` is equal to the game time. It sets the wind power to 0 whenever the game time is not inbetween a wind event.
The wind power rises linearily to its peak (`PMax * windScalingFactor`) at `middleTime` (time in the middle between `windEventStartingTime` and `windEventEndingTime`).
Afterwards, the wind power decreases linearily until it is 0 at `windEventEndingTime`.

---

dynamicPhotovoltaicChange

```
public void dynamicPhotovoltaicChange()
```

Input Arguments

None

Description

This function “simulates” the sun of the model. The sun is assumed to rise at 06:00, it linearily rises to the peak value at 12:00 and it linearily descends again until it sets at 18:00.
The amount of power generated by the sun can be controlled by `solarScalingFactor` which can be set in the level description.
This is useful if bad weather conditions are to be simulated. The scaling factor is between 0 and 1 and is multiplied to the current power of the PV module.

The functions are defined as linear interpolations from 0 to `PMin` of the PV Modules, depending on the time from 06:00-12:00 and 12:00-18:00. It is 0 otherwise.

---

updateBatteries

```
public void updateBatteries()
```

Input Arguments

None

Description

This method is called in each frame of the game. It treats the batteries differently depending on if they are manually operated and what Bus Type they are attached to.

The different cases are:

1. **Not interactive & Type "P" or "W"**

   If bus is generating power and the battery is not full -> Charge it.
   If the bus is not generating any power and there already is a bit of charge -> Discharge it.
   If the battery is at full capacity or it is empty -> Set it to idle.

2. **Not interactive & Type "C"**

   If the slack bus exports more than 0.5 & the battery is not full & it isn't already charging & subtracting the battery power from the slack bus export does not make it import -> Charge it.
   If the slack bus imports more than 0.5 & the battery is not empty & it isn't discharging & adding the battery power to slack bus export does not make it export -> Discharge it
   If the battery is full or empty -> Set it to idle.

3. **Interactive battery**

   If the battery is charging and the power of the bus goes to 0 -> Set it to idle
   If the battery is empty -> Set it to idle and set the bus power back to normal.
   If the battery is full -> Set it to idle and set the bus power back to normal.
   If the user has set the battery to idle -> Set the bus power back to normal.
   If the user has set the battery to charging -> Add the power use of the

battery to the bus power. (Bus uses more power to charge battery)
If the user has set the battery to discharge -> Subtract the power use of the battery to the bus power. (Bus generates more power from battery)

---

modifyBatteryState

```
public void modifyBatteryState(int batteryID, String newState)
```

Input Arguments

- `batteryID`: The ID of the battery to be changed.
- `newState`: The new State in form of a string.

Description

This method handles all user interaction with the batteries. The user can click the UI Buttons provided by the game class to change the charging state of the battery.

The switch statement identifies the string passed to the method.
The three options are:

- "Charge": Sets it to charging.
- "Discharge": Sets it to discharge.
- "Idle": Sets it to idle.

The method then iterates over every battery in the level and if it finds the one matching the ID given, it makes a copy of the bus associated to it.
To be able to change the state one of the following requirements has to be met:

- To charge it, the power of the battery added to the bus must not exceed `PMax`. This is to make sure the power drain does not go above limits.
- To discharge it, the power of the battery subtracted to the bus must not be lower than `PMin`. This is so that it doesn't generate more than the bus can handle. e.g. Consumers can't start producing power.
- Setting the battery to idle is always allowed.

---

isActiveBatteryChangeable

```
public bool isActiveBatteryChangeable()
```

Input Arguments

None

Returns

A boolean that is `true` if the battery is interactive. (if `AllowsChange` of `activeBattery` == `true`)

Description

This function tells the game class whether the selected battery is interactive. If it is then it will extend the User Interface to accomodate for the buttons.

---

manualChangeBusPower

```
public void manualChangeBusPower(double amount)
```

Input Arguments

- `amount`: The amount of change the user wishes to change the bus. This is `0.01f` in the game class.

Description

If the bus allows manual change, the bus power will be adjusted by `0.01f`.
In case the bus is a generator, the sign of `amount` is flipped. This is done because otherwise to increase power generation, one would have to press the down key which is not as intuitive.
The `PForBattery` is changed aswell, as it is the reference point to which battery power is added or subtracted from.
The model is then flagged as not solved to show the change in power.

---

toggleLineText

```
public void toggleLineText()
```

Input Arguments

None

Description

This function toggles the visibility of the text on the powerlines.

---

increaseLinePower

```
public void increaseLinePower()
```

Input Arguments

None.

Description

This method increases the maximal power of all lines.

---

containsBatteries

```
public bool containsBatteries()
```

Input Arguments

None

Returns

A boolean which is `true` if there are batteries in the power model, `false` otherwise.

Description

This function returns `true` if there are batteries in the power model.

---

losingCondition

```
public Boolean losingCondition()
```

Input Arguments

None

Returns

A boolean which is `true` if one of the lines in the power model has a power flowing through it that exceeds `line.PMax`, `false` otherwise.

Description

This method gets called in every frame of the game class. In case it returns `true`, the game class starts the countdown for a system overload. If it reaches 0, the player loses.

---

winCondition

```
public Boolean winCondition()
```

Input Arguments

None

Returns

A boolean which is `true` if `timeInLevelHour` is greater than `24`. This means the player has passed a whole day without the system breaking.

Description

This method is called in every frame of the game class. If it returns `true` the game class will display the winning screen and the player can then select a new level.

Update

```
public void Update(MouseState mouseState, float elapsedTime)
```

Input Arguments

- `mouseState`: The current state of the mouse, containing the position, button clicks and others.
- `elapsedTime`: The current elapsed time in seconds of the whole program. It starts as soon as the program starts.

Description

This method is responsible for handling all game logic of the power model.
It keeps track of the current time in minutes of the level by subtracting `startTime`, which is the time in seconds when the level was started, from `elapsedTime` which is the current time in seconds since the program has started.
It then calculates the current time in hours by dividing the time in minutes by 60 minutes.
Then it loads any possible events that might occur at that time.
Then it handles the change in sun intensity and wind.
In the end it calls `updateBatteries()`, calls the Update function of all `lines` and calls `solve()` to solve the model if necessary.

---

checkMouse

```
public void checkMouse(MouseState mouseState)
```

Input Arguments

- `mouseState`: The current state of the mouse including position and mouseclicks.

Description

This function sets the UI focus on a `Bus` and/or `Battery` if the player clicks on it.
If a left click is registered, it sets the flag `mouseClicked` to `true` to prevent continous execution of the function.
Then if a bus or battery `Rectangle` contains the position of the mouse during the click, `activeBus` or `activeBattery` is set to the corresponding bus/battery ID.
If the player clicks on the UI to change a battery state, the bus or battery shouldn't be deselected.
If the player clicked empty space, the activeBus gets set to the slackbus and the active battery is set to `NONE` which is 999, signifying no battery is chosen.
This is done so that the game class can check for `NONE` and hide the battery panel if none is selected.

---

getActiveBatteryPosition

```
public Vector2 getActiveBatteryPosition()
```

Input Arguments

None

Returns

A `Vector2` containing the position of the active battery.

Description

This function can be used to fetch the position of the current active battery.

---

drawBatteryInfo

```
public void drawBatteryInfo(SpriteBatch spriteBatch, SpriteFont theText)
```

Input Arguments

- `spriteBatch`: The spriteBatch used to draw the information.
- `theText`: The spritefont used to draw the text.

Description

This method displays the text on the battery panel in the game window.
If the battery is active, it will display "Active Battery:" otherwise it will display "Passive Battery".
If there is a battery selected, it will iterate over all available batteries and when it finds the one matching `activeBattery` it will print the relative information.

---

getActiveBusPosition

```
public Vector2 getActiveBusPosition()
```

Input Arguments

None

Returns

A `Vector2` containing the position of the active bus.

Description

This function can be used to fetch the position of the current active Bus.

---

drawBusInfo

```
public void drawBusInfo(SpriteBatch spriteBatch, SpriteFont theText)
```

Input Arguments

- `spriteBatch`: The spritebatch used to draw the information.
- `theText`: The spritefont used to draw the information of the bus.

Description

This method will print the text of the bus information panel on the screen.
It will print the currently active bus ID and then uses the `toString()` function of the active bus to draw the relevant information.

Draw

```
public void Draw(SpriteBatch spriteBatch, SpriteFont theText)
```

Input Arguments

- `spriteBatch`: The spritebatch used to draw all the sprites.
- `theText`: The spritefont used to draw all the text.

Description

This method is responsible for drawing all elements contained in the power model.
It is called once every frame.
First it will draw the background of the current level.
Then it iterates through all lines, buses and batteries to draw their graphics.
And finally it draws a string representing the current time on the bar on the bottom of the window.

---

toString

```
public String toString()
```

Input Arguments

None

Description

This method can be used for debugging purposes to show which elements have been added to the model.