Light Simulation

A simple procedural light simulation for point lights and rays. Customize different maps, walls, and lights, with the ability to save as a .txt or interact realtime

Description

Each light is composed of multiple rays, which are drawn from the center point in varying directions. Each ray continues until it reaches the edge of the map, a wall, or the light's maximum strength.

Documentation

InputManager

The InputManager class handles all user input It creates a Map object, and prompts the user to customize the map or add lights It includes methods to determine usability of user input, and improves readability of main method

- `getMapInput`
  - Prompts user for map dimensions, and constructs map accordingly
- `getWallInput`
  - Prompts user for two points to add a in between on the map
  - Loops until user decides to finish
- `runRealTime`
  - Asks user if they want to run project in real time, runs the point or ray demo respectively. Otherwise, returns false
- `getLightsAndSimulate`
  - If the user doesn't want to run realtime, they are prompted too add lights
  - Function loops until the user decides to finish
  - Once finished, the function prints out the output
- `returnAsFile`
  - returns simulated map as txt file, if user requests
- `runPointDemo (int strength, int lightComplexity)`
  - creates a JFrame object that tracks mouse x/y position and translates screenspace relative position into arrayspace relative position for the source of a pointLight
  - transfers the grid to an html table inside the JFrame Object (this can probably be optimized)

    - takes in a strength and light complexity parameter for the pointLight

- `runRayDemo (int angle, int width)`
    - creates a JFrame and directional ray object at 0, 0 which tracks mouse x/y position and determines angle relative to screenspace origin. It sets a directional ray's angle to angle determined
    - transfers the grid to an html table inside the JFrame Object (this can probably be optimized)
    - takes in a width and strength parameter for the directionalRay object

- `countSpaces`
    - counts spaces in given string, used to verify input validity

- `isNumbersAndSpaces`
    - ensures each character in given string is a valid input character

- If a realtime demo isn't selected, the user is prompted to add lights, and is then given the simulated frame, with the option to save to a .txt file.

Map

The Map Class contains a 2d array of `Tile` objects, representing the grid. It's constructed with a length and width, to determine the grid's dimensions, with a default size of 101 by 101 `Tile` objects.

Also contains an arrayList `lights` of `Light` objects, which can all be simulated on the grid through `simulate`

- `drawLine (int x1, int y1, int x2, int y2)`
    - draws a wall based on given input points 1 and 2, returns false if the points given are not possible

- `addLight (Light light)`
    - adds a light to the `lights` arrayList

- `clear`
    - clears all `Tile` objects in grid that have a light-related state

- `clearLights`
    - removes all lights in the `lights` arrayList

- `simulate`
    - Invokes the `simulate` method on each `Light` inside the `lights` arrayList onto the Map's grid

- `toString`
    - converts the grid to a String, by getting each `Tile` objects String value

- `deleteString`

    - returns a string of unicode backspace characters "\u0008" based on the grid size
    - used to clear console during testing, however is not currently used

Tile

Used to more clearly organize the grid, as compared to raw strings. Has `state`, which is of an enum called `tileType`

- `enum tileType`
    - enum used to represent `state` of a `Tile` object
    - `EMPTY, HORIZONTAL_WALL, VERTICAL_WALL, SLANT_RIGHT_WALL, SLANT_LEFT_WALL, LIGHT, SOURCE, CENTER, NULL`

- `toString`
    - returns a single character string based on `state`

Light

Parent class of `PointLight` and `DirectionalRay`. Stores important information, such as x/y position, strength, and parent map. Also creates template method `simulate` to be overwritten (necessary since both `PointLight` and `DirectionalRays` can be stored and must be simulated from within `lights` arrayList of type `Light` in `Map`). Otherwise, contains necessary `get` and `set` methods.

PointLight

Extends `Light` Creates a circular light object by sending rays at different angles across 360 degrees. The constructor takes in an additional parameter, `lightComplexity` which determines the number of rays drawn across 360 degrees, by default is 36 (a ray will be drawn each 10 degrees). The `lightComplexity` dictates the values of the `angles` array, which saves each angle to be simulated (calculated through `findRayDegrees`) The `angles` array ranges from 90 to -90, since the `drawRay` function draws in both directions

- `simulate (Map m)`
    - takes in Map param to simulate on, defaults to map the object was constructed with if it wasn't given
    - loops through the `angles` array and invokes `drawRay` for each angle

- `drawRay (int angle, int x, int y, int grid)`
    - draws a ray from the given x and y values on the given grid (which is redundantly input and output but no matter)
    - increments x from starting point, calcuating y using `Math.tan`
    - then decrements x from starting point, calculating y the same way
    - the ray continues until the length of the ray reaches `strength` or until it hits an edge or tile with any variation of the `WALL` state
    - the drawRay function is also used in `DirectionalRay` which is why the x and y values are a part of the input (although technically redundant due to the single-origin-point nature of `PointLight`)

- Also contains standard `get` and `set` methods, with additional methods to convert between array coordinates and cartesian ones (necessary for mathematical calculates inside drawRay)

DirectionalRay

Extends `Light` Creates a beam of light by sending rays at the same angle, but from offset light sources It's constructed with two new parameters, `angle` and `width` which define accordingly. These parameters also define a 2D int array `sources` which saves the coordinate positions of each light source. The arrangment of points in `sources` is either 0, 45, 90, 135, or 180 degrees (to prevent massive skewage on steeper input angles). The points aim to be as perpendicular to the beam's angle as possible, for aesthetic.

- `simulate`
  - takes in Map param to simulate on, defaults to map the object was constructed with if it wasn't given
  - loops through each point in `sources` and invokes `drawRay`
- `drawRay (int angle, int x, int y, int grid)`
  - draws a ray from the given x and y values on the given grid (which is redundantly input and output but no matter)
  - increments x from starting point, calcuating y using `Math.tan`
  - then decrements x from starting point, calculating y the same way
  - the ray continues until the length of the ray reaches `strength` or until it hits an edge or tile with any variation of the `WALL` state

Authors

Jeremy Flint [@jzfcoder (https://github.com/jzfcoder)](https://github.com/jzfcoder)