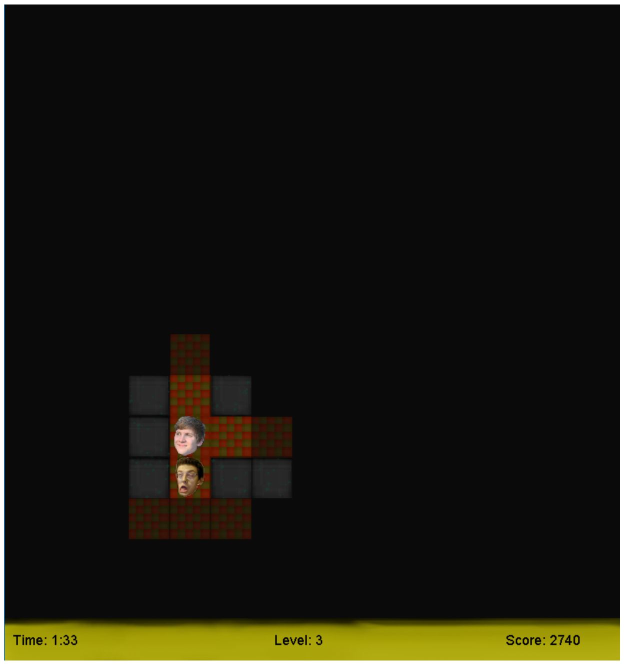
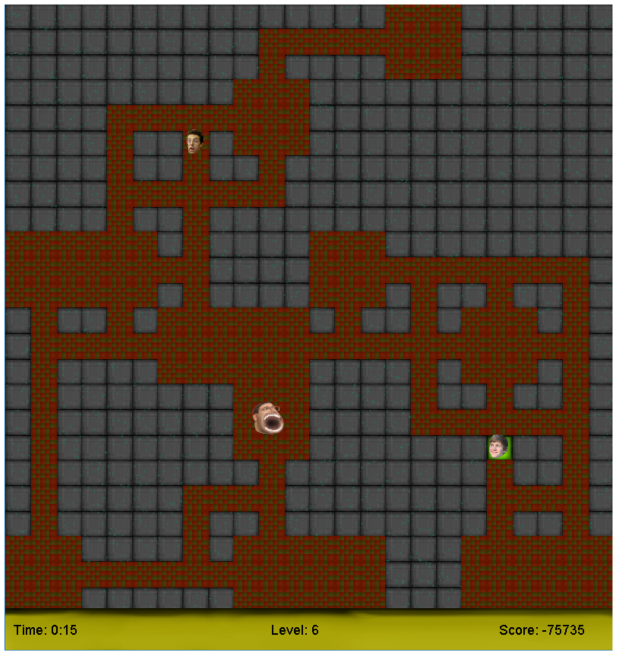
DUNGEON CRAWL Project Program Specification

by Ronin Rodkey and Will Christians



Screenshot from 2-minute mode



Screenshot of standing on a Light Subtile

General Description:

DUNGEON CRAWL is a game made up of tiles in which the player explores a ‘dungeon’ to get to the next level. While exploring, the user is aware of their points, the level they’re on, and the time remaining (in 2 minute mode) or elapsed (in endless mode).

The user moves the player with WASD around the navigable portions of the dungeon. The player’s sprite has limited vision, meaning it can’t see the whole map at once unless its on a light tile. While the player moves, the chaser pursues, following the player’s exact movements. If the chaser occupies the same subtile as the player, the chaser takes the player’s flashlight, and the player loses points.

In the 2-minute mode, the player has 120 seconds to get as far as possible and earn as many points as they can. The endless mode has no goal, but lets the user see what level they can get to before they are driven insane by the darkness (in real life).

The game also features a leaderboard system which allows the player to keep track of their scores.

Program Breakdown

Classes:

MazeGame

The MazeGame class is where the magic happens. Everything leads back to this class. This class managing the game at a large scale (levels, score, time, etc.), while also calling a lot of methods in other class to draw, move, or construct.

MainMenu

The MainMenu class manages everything that isn't 'in game.' It manages drawing the menu screen, the how to play screen, and the leaderboard screen. It communicates with the MazeGame class when the user wants to start a game.

Map

The Map class is where the game's map is managed. This class holds the layout of the dungeon via a Tile matrix, and does all the heavy lifting required to make the game actually work.

Tile

The Tile class embodies the little pieces of the map, each tile has a type and an array of subtiles which the player moves around in. Many of these exist in the map, but only 1/9th of the subtiles.

Subtile

The smallest part on the map, the Subtile is part of a tile. The subtile allows us to manage functions like player blindness, movement, and design at a more precise level. Many of these exist in one map.

Sprite

An abstract class for the Player and Chaser. The one functional method in this class turns character input into movement. This class is also responsible for drawing and calculating the amount of darkness surrounding the player.

Player

Where information about the Player is stored. It calls the Sprite's move class to move once it receives character input from the keyTyped method in the MazeGame class.

Chaser

Where information about the Chaser is stored. It has a queue of characters that is added to whenever the player moves, then follows the movements via the update method. The update method is called ever so often by the Runner's run class depending on the chaserspeed variable in the MazeGame class.

Runner
Has one method which loops in a thread of its own, allowing the program to repeatedly draw the pictures and keep track of game data.

-- Data Structures --

Queue/Node
Has a starting node and, when new things are added, they are added to the back (FIFO). Used by the Chaser to make it move correctly.

TableList/Pair
A data structure that is basically just an ArrayList of two variables that can be sorted by one of the two variables.

Member Fields

Class	Field	Description

`MazeGame`	`public static final int WIDTH`
	`public static final int HEIGHT`
		The dimensions of the window, used in the constructor. Set as 800 and 850, respectively.

`MazeGame`	`public static final int FPS`	The frames per second the game runs at.
`MazeGame`	`public static Map map`	The map the game loads, updated when the game starts and in the levelup method.
`MazeGame`	`public static Player player`	The player in the map. Used when calling the move method of the player class inside the keyTyped method.
`MazeGame`	`public static Random rand`	A random number generator used to generate the map's seeds.
`MazeGame`	`public static JFrame frame`	The JFrame window that the game is in. It's helpful to have it in the class so that methods can call functions on it without having to take it as an input.
`MazeGame`	`public double time`	The time elapsed/remaining depending on the game mode, in seconds. Accessed in the map class when drawing the info.
`MazeGame`	`public static double timechange`	How much time increases/decreases each frame. Used in the run method in the runner class.
`MazeGame`	`private static Scanner keyboard`	Used to take input for the addToLeaderboard method.
`MazeGame`	`private static int chaserspeed`	How many frames between the chaser's updates. Decreases whenever the player levels up.

MazeGame	`public static int score`	How many points have been scored. Increases based on time passed and levels. Decreases whenever the player touches the chaser.

MainMenu	`public static final int WIDTH`
	`public static final int HEIGHT`
		The dimensions of the window, used in the constructor. Set as 800 and 850, respectively.

MainMenu	`public char currentScreen`	The current screen being displayed. ‘m’ for main menu, ‘l’ for leaderboards, ‘h’ for how to play, ‘o’ for game over. Used in a switch statement inside the draw class.
Map	`public Tile[][] grid`	The matrix of tiles that make up the map. Generated by the build method.
Map	`public int size`	The size of the map (side length of the tile matrix). Increases when leveling up.
Map	`public long seed`	The seed used to build the levels. Semi-obsolete, but useful for debugging.
Map	`private Random rand`	The random number generator used to generate the map.

Map	`public int HEIGHT`
	`public int WIDTH`
		The dimensions of the window, used in the constructor. Set as 800 and 850, respectively. Useful for graphical calculations.

Map	`public Player player`	The player inside the map.
Map	`public static JFrame frame`	The JFrame window that the game is in. It’s helpful to have it in the class so that methods can call functions on it without having to take it as an input.

Tile	`public int xPos;`
	`public int yPos;`
		The x positions and y positions of the tile inside the map’s tile matrix.

Tile	`public int type;`	The type of tile (0 = walls, 1 = room, 2 = hallway, -1 = uninitialized)

Tile	`public Tile north;`
	`public Tile south;`
	`public Tile east;`
	`public Tile west;`
		Pointers to the surrounding tiles. Needed for proper display of the hallways.

Tile	`public static JFrame frame`	The JFrame window that the game is in. It’s helpful to have it in the class so that methods can call functions on it without having to take it as an input.
Tile	`public Subtile[][] subtiles`	A 3x3 matrix of smaller tiles which are toggled on and off depending on whether or not the player can move into them.
Subtile	`public boolean show`	The toggle mentioned in the description for subtiles
Subtile	`public Tile parent`	The tile in which the subtile is a part of.
Subtile	`public boolean isGoal`	A toggle that determines whether or not the subtile is a goal tile. There is only one goal subtile per map.

`Subtile`	`public Color moss`	Color for moss dispersed on the bricks
`Subtile`	`public boolean isLS`	True if it's a "light" subtile
`Subtile`	`Random rand`	Uses to randomly generate elements in the subtile class.
`Subtile`	`public int blindness`	0 if we see the subtile perfectly, 1 if it's dim, 2 if we can't see it.
`Subtile`	`public boolean isLS`	A boolean that determine whether or not the subtile is a light tile (the green one that shows the whole map).

`Sprite`	`public int xCoord`
	`public int yCoord`
		The coordinates of the tile in which the player is located.

`Sprite`	`public int xSubCoord`
	`public int ySubCoord`
		The coordinates of the subtile in which the player is located.

`Sprite`	`public Map m`	The map in which the player lives.

`Sprite`	`public Color dim`
	`public Color blind`
		Colors for blindness

`Player`	`public Chaser chaser`	The chaser which follows the player's movements.
`Chaser`	`public Queue<Character> moveList`	A list of moves in order, so that the chaser can exactly replicate the player's movement. Each time the player moves, their movement is added to the queue.
`Chaser`	`public boolean doMove`	Whether or not the chaser should move when it updates. If the player hasn't moved yet, this is false.
`Queue`	`public Node<T> start`	The first node in the queue.
`Node`	`public E elem`	The element held by the node.
`Node`	`public Node<E> next`	The node 'behind' this node in the queue.
`TableList`	`private ArrayList<Pair> list`	Holds the pairs. TableList was implemented so that the pairs could be sorted so the leaderboards were ranked correctly.

`Pair`	`public String x`
	`public int y`
		A name (x) and a score (y) held by the pair.

Methods

All methods are private by default, only public if called by another class.

Class	Method Header	Description

`MazeGame`	`public void keyPressed(KeyEvent e)`
	`public void keyReleased(KeyEvent e)`
	`public void keyTyped(KeyEvent e)`
	`public void addNotify()`
		All methods used when implementing keyListener so the program can read input and move the player sprite. keyTyped is the method that actually moves the player sprite.

`MazeGame`	`public static void main(String[] args)`	Displays the main menu and makes the graphics

		window work.
`MazeGame`	`public static void startGame(char enterCode)`	Starts the game with an enter code that tells the length of the game (timedn or endless). It needs to be static because its called static-ly in the MainMenu class.
`MazeGame`	`public void paintComponent(Graphics g)`	Sets the background then paints the map. Needed for graphics.
`MazeGame`	`public static void levelUp()`	Sets the map to a new map that is a little bigger. Called staticly in the Player class whenever the player moves into the goal.
`MazeGame`	`private static void addToLeaderboard(int points)`	Asks the user's name via console then writes their score and name to the leaderboard txt file. Called when game ends.
`Runner`	`public void run()`	Repeatedly called to update the graphics, increase the timer, and move the chaser.

`MainMenu`	`public void keyPressed(KeyEvent e)`
	`public void keyReleased(KeyEvent e)`
	`public void keyTyped(KeyEvent e)`
	`public void addNotify()`
		Used to read input from the user, calls startGame with the enter code (1,2,3,etc.). keyPressed is the one that actually does something.

`MainMenu`	`public static void main(String[] args)`	Sets up the graphics for the main menu.
`MainMenu`	`public void paintComponent(Graphics g)`	Displays the picture of the main menu using ImageIO.
`MainMenu`	`private void printLeaderboard(Graphics g)`	Prints the leaderboard.txt file on the screen.
`Map`	`private void build()`	Initializes all the tiles in the map by calling the generate method in the tile class, pruning dead-end hallways, and constructing the player. Instance method because it uses instance variables.
`Map`	`private void genGoal()`	Starting with the top left tile and moving right then down, probabilistically determines where the goal will be. Calls itself if it gets through the loop without setting any tile as a goal. Instance method because it uses instance variables.
`Map`	`private void genLumTiles(int count)`	Generates the light tiles in the map recursively. Called by the build method.
`Map`	`public void draw(Graphics g, double time)`	Draws the map in the graphics. Uses the double value of time when called to print the time at the bottom with the printInfo method. Instance method because it uses instance variables.
`Map`	`private void pruneHallways()`	Finds dead-end hallways using the neighbors method in the Tile class, then sets them to be empty. Repeatedly called by the build method. Instance method because it uses instance variables.
`Map`	`private void print()`	Prints the map, not currently used but helpful when debugging graphics. Instance method

		because it uses instance variables.
Map	`private void drawInfo(double t, Graphics g)`	Called by the map's build class to print the panel at the bottom that displays the game's info. Instance method because it uses instance variables.
Tile	`public void generate(Random rand, boolean start)`	Called by the map's build class and recursively calls itself to generate. If it is the start the boolean will be true, setting the tile as a room and generating neighbors. If it generates a non-empty square, it generates its neighbors that haven't already been initialized. The random parameter allows the map to have a seed that generates everything the same time. Instance method because it uses instance variables.
Tile	`public void initializeSubtiles()`	Each tile is a 3x3 grid, this sets the pattern of the 3x3 grid so that the pattern looks correct. Called by the build method in the map class. Instance method because it uses instance variables.
Tile	`public void draw(Graphics g, int size, int W, int H)`	Draws the tile by loading images then drawing all the subtiles of the tile, called by the Map's draw method. Instance method because it uses instance variables.
Tile	`public int neighbors()`	Counts the neighbors to determine whether or not the tile is a dead end (1 neighbor). Used in the pruneHallways method in the Map class. Instance method because it uses instance variables.
Subtile	`public void drawSub(Graphics g, int size, int W, int H, BufferedImage brick, BufferedImage c1, BufferedImage c2, JFrame frame)`	Draws the subtile in the proper location on the screen. Uses parameters to calculate where to print it and the size with which to print it. Called by the draw method in the Tile class. Instance method because it uses instance variables.
Subtile	`public void makeGoal()`	Sets the subtile to be the goal. Called by the genGoal method in the Map class. Uses instance variables.
Subtile	`private void drawCarpet(int xPix, int yPix, int width, BufferedImage c1, BufferedImage c2, JFrame frame, Graphics g)`	Draws a carpet of specified width at point (xPix, yPix). C1 and c2 are the images which are used to make up the individual squares of the carpet.
Subtile	`private BufferedImage changeCarpet(BufferedImage carpet, BufferedImage c1, BufferedImage c2)`	If "carpet" is c1, change it to c2 and vice-versa. Used in drawCarpet.
Sprite	`public void moveSprite(char c)`	Takes character input and tries to move the sprite (chaser or player) to the position. It checks if the spot the sprite will be moving into is part of the floor, and moves the sprite if it is.
Sprite	`public abstract void draw(Graphics g, JFrame frame)`	Ensures all sprites have a draw method. Not necessary for anything, but it's abstract and cool, unlike those OTHER methods.
Player	`public boolean isAtLS()`	Returns true if we're at a "light" subtile

`Player`	`public void draw(Graphics g, JFrame frame)`	Draws the player via imageio in the correct position on the screen.
`Player`	`public void initBlindness(int width)`	Sets the level of blindness (or smoke, light, what have you) for each subtile based on the player's position. Called from the map's draw such that when we go to draw the individual tiles of the maze, we only draw what the player will be seeing.
`Player`	`private void drawBlindness(graphics g, int width)`	Draws blindness (or smoke, light, what have you) based on the values assigned to each subtile in prior methods.
`Player`	`public boolean isAtLS()`	Return true if the player is on a light tile, if not, then returns false.
`Player`	`public void move(char c)`	Calls the moveSprite method in Sprite. Also checks to see if the tile the player is moving into is a goal or light tile and does the appropriate actions.
`Chaser`	`public void draw(Graphics g, JFrame frame)`	Draws the chaser via imageio in the correct position on the correct subtile.
`Chaser`	`public void update()`	Called every so often by the run class in Runner. Basically calls moveSprite with the character that is first in the queue. If the chaser doesn't actually move (if it hits a wall), then it tries to move again.
`Chaser`	`public Subtile getChaser()`	Returns the subtile the chaser is on
`Queue`	`public void add(T elem)`	Adds an element of type T to the back of the queue.
`Queue`	`public T pop()`	Removes the element at the front of the queue, then returns it.
`TableList`	`public void add(String x, int y)`	Adds a pair to a table list. Used for making the leaderboard work.
`TableList`	`public void sort()`	Sorts the TableList from highest score to lowest score.
`TableList`	`public Pair get(int i)`	Gets the ith element in the table list.
`TableList`	`public int size()`	Returns the size of the table list.

Constructors

Class	Constructor Header	Description
`MazeGame`	`public MazeGame()`	Sets dimensions, starts keyListener, establishes runner thread.
`MainMenu`	`public MainMenu()`	Sets dimensions, starts keyListener.
`Map`	`public Map(int size, long seed, int W, int`	Sets member fields, then calls the build method

	`H, JFrame frame)`	which constructs the map's layout tile-by-tile.
`Tile`	`public Tile(int x,int y,int t, JFrame frame)`	Sets a bunch of member fields.
`Subtile`	`public Subtile(Tile t, int x, int y,boolean b, JFrame frame)`	Sets a bunch of member fields.
`Player`	`public Player(Map m, int x, int y, int xSub, int ySub)`	Sets a bunch of member fields.
`Chaser`	`public Chaser(Map m, int x, int y, int xSub, int ySub)`	Sets a bunch of member fields.
`Node`	`public Node(E elem)`	Puts an element of type E into the node.
`Pair`	`public Pair(String s, int i)`	Puts a string and an int into the Pair.

How it Works:

The bulk of the program surrounds the MazeGame class, which starts of by contacting the MainMenu class which learns what the user wants to do. After the user has selected one of the game options, the MainMenu class sends the request over to the startGame method in the MazeGame class.

When a game is started, a Map is constructed which involves creating an array of tiles, making the bottom middle one a room, then calling the generate method inside the Tile class. The generate method randomly sets the room's type (wall, room, or hallway) and then calls the same generate method on its uninitialized cardinal neighbors (north, south, east, west). After all the room types are set, the subtiles are initialized through the initializeSubtiles method in the Tile class, which correctly forms the hallways and rooms in a 3x3 matrix of subtiles. After that, dead end hallways are cut off by the pruneHallways method (in Map), and a goal is set by the setGoal method (also in Map). Then the player and chaser is created and placed in the start tile. Back to the MazeGame class, everything is drawn.

The tiles are drawn in relation to the player with various brightness levels, and the chaser is drawn. While the game is running, time and score are being kept track of through member fields in the MazeGame method and are changed by the Runner class's run method.

The user can move the player with their keyboard's WASD keys using the keyListener interface. The keyTyped method in the MazeGame class tells the player to move, which moves the player on the screen then adds that movement to a queue, or list of movements the chaser will follow.

When the player reaches the goal, the levelUp method in the MazeGame class is called, and a new slightly bigger map is generated through the same process mentioned above. In the two minute mode when the time runs out, the JPanel closes and the console asks for the user's name for entry on the leaderboard.

The leaderboard functions by reading the leaderboard.txt file and putting its information into a data structure, a TableList, which piggybacks on an ArrayList to store pairs. We use this data structure to allow us to easily sort the list by score.