Clarifications

Project 2: CS61BYoG

This section will be for any details we believe were not clear enough in the original spec. It is placed at the top for better visibility.

- For phase 1, your game does not necessarily need to suppprt `playWithKeyboard()` but it must support a portion of the `playWithInputString()` method. Specifically, you should be able to play your game with an input string which starts a new game, types in a seed, begins the game/generates the world, and then returns the array representing the world at that point. This essentially means you must support any string of the format "N#######S" where each # is a digit and there can be an arbitrary number of #s.
- Letters in the input string can be of either case and your game should be able to accept either keypress (ie. "N" and "n" should both start a new game from the menu screen).
- In the case that a player attempts to load a game with no previous save, your game should end and the game window should close with no errors produced.
- In the base requirements, the command ":Q" should save and completely terminate the program. This means an input string that contains ":Q" should not have any more characters after it and loading a game would require the program to be run again with an input string starting with "L"
- Your game should be able to handle any positive seed up to 9,223,372,036,854,775,807.
- Your game should **NOT** render any tiles or play any sound when played with `playWithInputString()`.
- StdDraw does not support key combinations. When we say ":Q", we mean ":" followed by "Q".
- Your project should only use standard java libraries (imported from java.*) or any libraries we provided with your repo. This is only

relevant to the autograder so if you'd like to other libraries for gold points and for the video, feel free to do so.
- Any TETile objects you create should be given a unique character that other tile's do not use. Even if you are using your own images for rendering the tile, each TETile should still have its own character representation.
- The only files you may create must have the suffix ".txt" (for example "savefile.txt"). You will get autograder issues if you do not do this.

Introduction

In Project 2, you will create an engine for generating explorable worlds, which for lack of a better word we will call a "game". This is a large design project that will require you and one partner to work through every stage of development from ideation to presentation. The goal of this project is to teach you how to handle a larger piece of code with little starter code in the hopes of emulating something like a product development cycle. In accordance with this, the grading of this project will be different from other projects. Since there is no notion of "the correct answer" when it comes to game design and implementation, you will be assessed much like a performance review you might receive at an internship or job in addition to a very general autograder. While this means you will be graded slightly subjectively, we promise to be pretty nice bosses and will respect you as any boss should respect their hard working employees. Please talk to us if you feel the grading scheme feels unfair.

A video playlist discussing tips for working on this project can be found at this link. Slides for these videos can be found at this link.

There are several key deadlines for this assignment:

- Phase 0: Team formation: Due with lab on Friday February 16th.
  - Partnership Form
  - Solo Form for if you really want to work alone, but this is not recommended.
- Phase 1: World Generation: Due on gradescope by 2/26 at 11:59 PM.

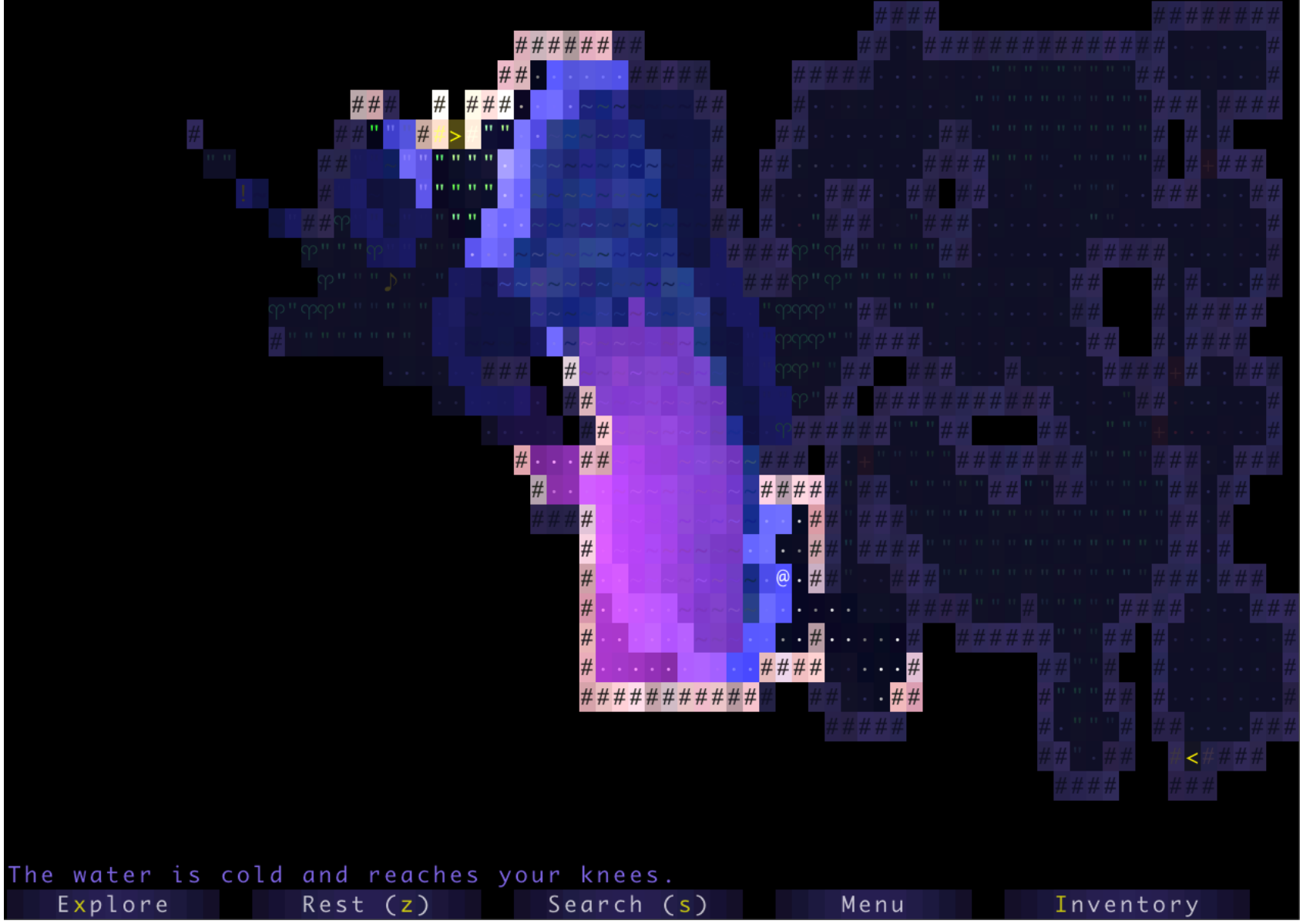
Finish by 2/23 for 6 points of extra credit.
- Phase 2: Playable Game: Due on gradescope by 3/5 at 11:59 PM.
- Lab Demo: Demo your project during lab on 3/7, 3/8, or 3/9. Sign-ups for demo slots will be released later.
- Gold Points: Add creative mechanics to your game, and create a public youtube video showcasing what you built by 3/9.

Now on to the assignment spec!

Overview

Your task for the next 3 weeks is to design and implement a 2D tile-based game. By "tile-based", we mean the world for your game will consist of a 2D grid of tiles. By "game" we mean that the player will be able to walk around and interact with the world. Your game will have an overhead perspective. As an example of a much more sophisticated game than you will build, the NES game "Zelda II" is (sometimes) a tile based overhead game:

The game you build can either use graphical tiles (as shown above), or it can use text based tiles, like the game shown below:



We will provide a tile rendering engine, a small set of starter tiles, and the headers for a few required methods that must be implemented for your game and that will be used by the autograder. The project will have two major deadlines. By the first deadline, you should be able to generate random worlds that meet the criteria below. By the second deadline, a player should be able to explore and interact with the world.

The major goal of this project is to give you a chance to attempt to manage the enormous complexity that comes with building a large system. Be warned: The game you build probably isn't going to be fun! Three weeks is simply not enough time, particularly for novice programmers. However, we do hope you will find it to be a fulfilling project, and the worlds you generate might even be beautiful.

Skeleton Code Structure

The skeleton code contains two key packages that you'll be using: `byog.TileEngine` and `byog.Core`. `byog.TileEngine` provides some basic

methods for rendering, as well as basic code structure for tiles, and contains:

- `TERenderer.java` - contains rendering-related methods.
- `TETile.java` - the type used for representing tiles in the world.
- `Tileset.java` - a library of provided tiles.

IMPORTANT NOTE: Do NOT change TETile.java's `charcter` field or `character()` method as it may lead to bad autograder results.

The other package `byog.Core` contains everything unrelated to tiles. We recommend that you put all of your game code in this package, though this not required. The `byog.Core` package comes with the following classes:

- `RandomUtils.java` - Handy utility methods for doing randomness related things.
- `Main.java` - How the player starts the game. Reads command line arguments and calls the appropriate function in `Game.java`.
- `Game.java` - Contains the two methods that allow playing of your game.

`byog.Core.Game` provides two methods for playing your game. The first is `public TETile[][] playWithInputString(String input)`. This method takes as input a series of keyboard inputs, and returns a 2D TETile array representing the state of the universe after processing all the key presses provided in input (described below). The second is `public void playWithKeyboard()`. This method takes input from the keyboard, and draws the result of each keypress to the screen.

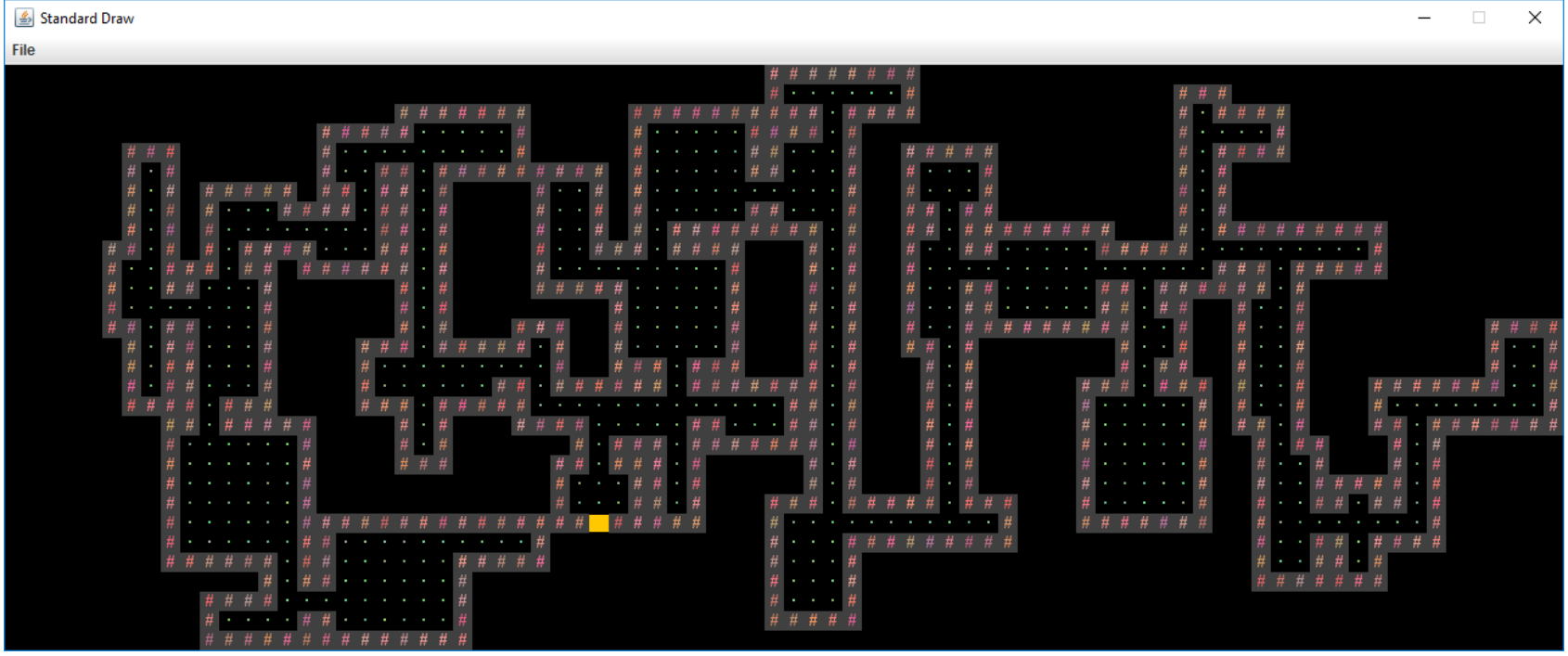
The game engine makes heavy use of `StdDraw`. You may need to consult the API specification for `StdDraw` at some points in the project, which can be found here.

Phase 1: World Generation

As mentioned above, the first goal of the project will be to write a world generator. The requirements for your world are listed below:

- The world must be a 2D grid, drawn using our tile engine. The tile engine is described in lab5.
- The world must be pseudorandomly generated. Pseudorandomness is discussed in lab 5.
- The generated world must include rooms and hallways, though it may also include outdoor spaces.
- At least some rooms should be rectangular, though you may support other shapes as well.
- Your game must be capable of generating hallways that include turns (or equivalently, straight hallways that intersect).
- The world should contain a random number of rooms and hallways.
- The locations of the rooms and hallways should be random.
- The width and height of rooms should be random.
- The length of hallways should be random.
- Rooms and hallways must have walls that are visually distinct from floors. Walls and floors should be visually distinct from unused spaces.
- Rooms and hallways should be connected, i.e. there should not be gaps in the floor between adjacent rooms or hallways.
- The world should be substantially different each time, i.e. you should not have the same basic layout with easily predictable features

As an example of a world that meets all of these requirements (click for higher resolution), see the image below. In this image, # represents walls, a dot represents floors, and there is also one golden colored wall segment that represents a locked door. All unused spaces are left blank.



Once you've completed lab 5, you can start working on your world generation algorithm without reading or understanding the rest of the spec.

It is very likely that you will end up throwing away your first world generation algorithm. This is normal! In real world systems, it is common to build several completely new versions before getting something you're happy with. The room generation algorithm above was my 3rd one, and was ultimately much simpler than either of my first two.

You're welcome to search the web for cool world generation algorithms. You should not copy and paste code from existing games, but you're welcome to draw inspiration from code on the web. **Make sure to cite your sources using @source tags.** You can also try playing existing 2D tile based games for inspiration. Brogue is an example of a particularly elegant, beautiful game. Dwarf Fortress is an example of an incredibly byzantine, absurdly complex world generation engine.

The Default Tileset and Tile Rendering Engine

The tile rendering engine we provide takes in a 2D array of `TETile` objects and draws it to the screen. Let's call this `TETile[][] world` for now. `world[0][0]` corresponds to the bottom left tile of the world. The first coordinate is the x coordinate, e.g. `world[9][0]` refers to the tile 9 spaces over to the right from the bottom left tile. The second coordinate is the y

coordinate, and the value increases as we move upwards, e.g. `world[0][5]` is 5 tiles up from the bottom left tile. All values should be non-null, i.e. make sure to fill them all in before calling `renderFrame`. **Make sure you understand the orientation of the world grid!** If you're unsure, write short sample programs that draw to the grid to deepen your understanding. **If you mix up x vs. y or up vs. down, you're going to have an incredibly confusing time debugging.**

We have provided a small set of default tiles in `Tileset.java` and these should serve as a good example of how to create `TETile` objects. We strongly recommend adding your own tiles as well.

The tile engine also supports graphical tiles! To use graphical tiles, simply provide the filename of the tile as the fifth argument to the `TETile` constructor. Images must be 16 x 16, and should ideally be in PNG format. There are a large number of open source tilesets available online for tile based games. Feel free to use these. Note: Your github accounts are set up to reject files other than .txt or .java files. We will not have access to your tiles when running your code. Make sure to keep your own copy of your project somewhere else other than Github if you want to keep a copy of your project with graphics for archival purposes. Graphical tiles are not required.

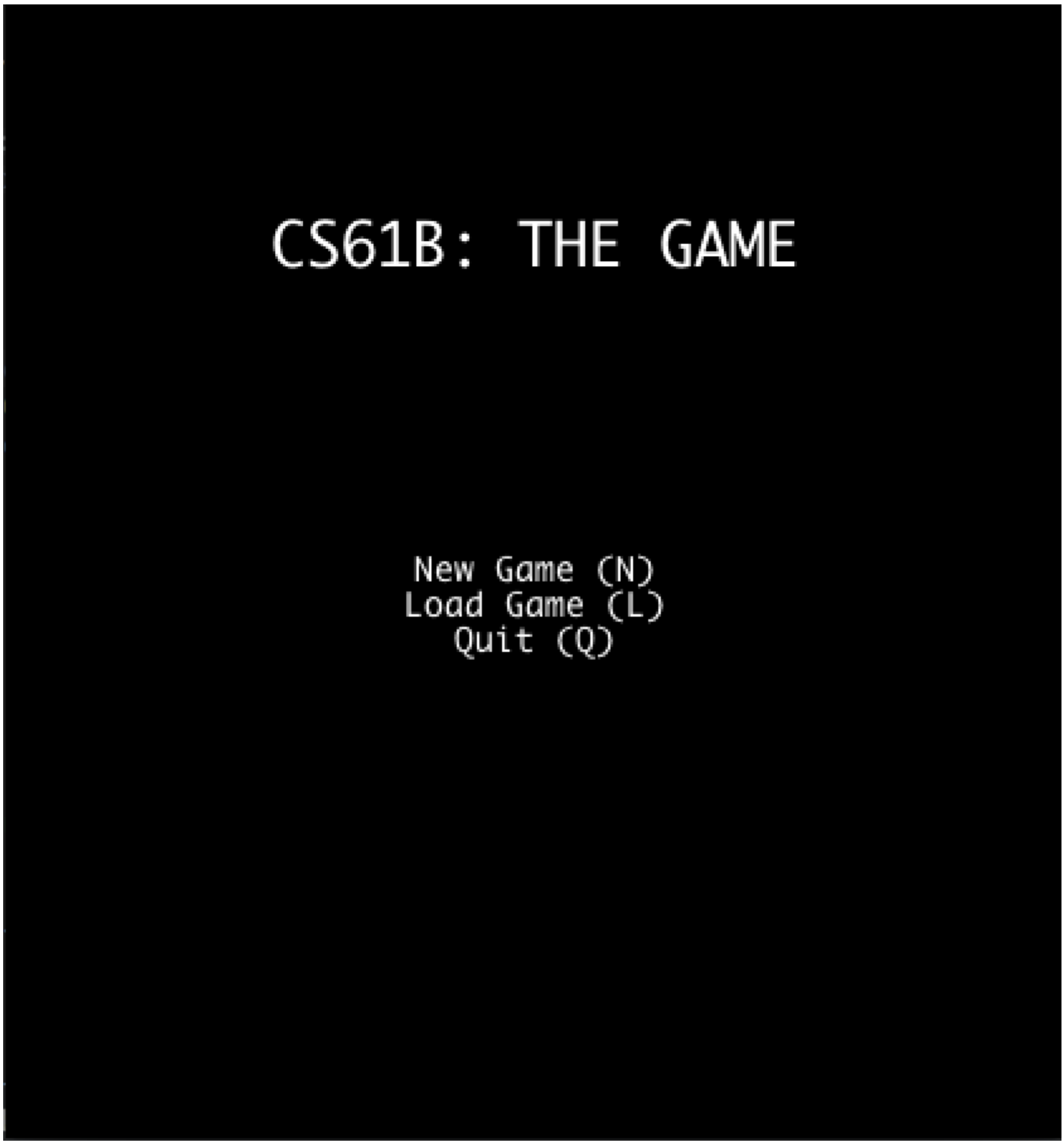
If you do not supply a filename or the file cannot be opened, then the tile engine will use the unicode character provided instead. This means that if someone else does not have the image file locally in the same location you specified, the game will still be playable but will use unicode characters instead of textures you chose.

The tile rendering engine relies on `StdDraw`. We recommend against using `StdDraw` commands like `setXScale` or `setYScale` unless you really know what you're doing, as you may considerably alter or damage the a e s t h e t i c of the game otherwise.

Starting the Game

Your game must support both methods of starting it, one using the `Core.Game.playWithKeyboard()` method, and the other using the `Core.Game.playWithInputSting(String s)` method.

When your `Core.Game.playWithKeyboard()` method is run, your game must display a Main Menu that provides at LEAST the option to start a new game, load a previously saved game, and quit the game. The Main Menu should be navigable only using the keyboard, using N for "new game", L for "load game", and Q for quit. You may include additional options.



After pressing N for “new game”, the user should be prompted to enter a “random seed”, which is an integer of their choosing. This integer will be used to generate the world randomly (as described later and in lab 5). After the user has pressed the final number in their seed, they should press S to tell the game that they’ve entered the entire seed that they want.

The behavior of the “Load” command is described elsewhere in this specification.

If the game instead started with `Core.Game.playWithInputString()`, no menu should be displayed and nothing should be drawn to the screen. The game should otherwise process the given String as if a human player was pressing the given keys using the `Core.Game.playWithKeyboard()` method. For example, if we call `Core.Game.playWithInputString("N3412S")`, the game should generate a world with seed 3412 and return the generated 2D tile array.

We recommend that you do not implement `Core.Game.playWithKeyboard()` until you get to phase 2 of the project (interactivity), though you’re welcome to do so at anytime. It will be easier to test drive and debug your world generator by using `playWithInputString` instead.

If you want to allow the user to have additional options, e.g. the ability to pick attributes of their character, specify world generation parameters, etc., you should create additional options. For example, you might add a fourth option “S” to the main menu for “select creature and start new game” if you want the user to be able to pick what sort of creature to play as. These additional options may have arbitrary behavior of your choosing. The behavior of N, L, and Q must be exactly as described in the spec!

Phase 2: Interactivity

In the second phase of the game, you’ll add the ability for the user to

actually play the game, and will also add user interface (UI) elements to your game to make it feel more immersive and informative.

The requirements for interactivity are as follows:

- The player must be able to control some sort of entity that can moved around using the W, A, S, and D keys. Lab 6 covers how to include interactivity in your game.
- The entity must be able to interact with the world in some way.
- Your game must be deterministic in that the same sequence of keypresses from the same seed must result in exactly the same behavior every time. It is OK if you use a pseudorandom generator, since the `Random` object is guaranteed to output the same random numbers every time.

Optionally, you may also include game mechanics that allow the player to win or lose (see gold points below). Aside from these feature requirements, there will be a few technical requirements for your game, described in more detail below.

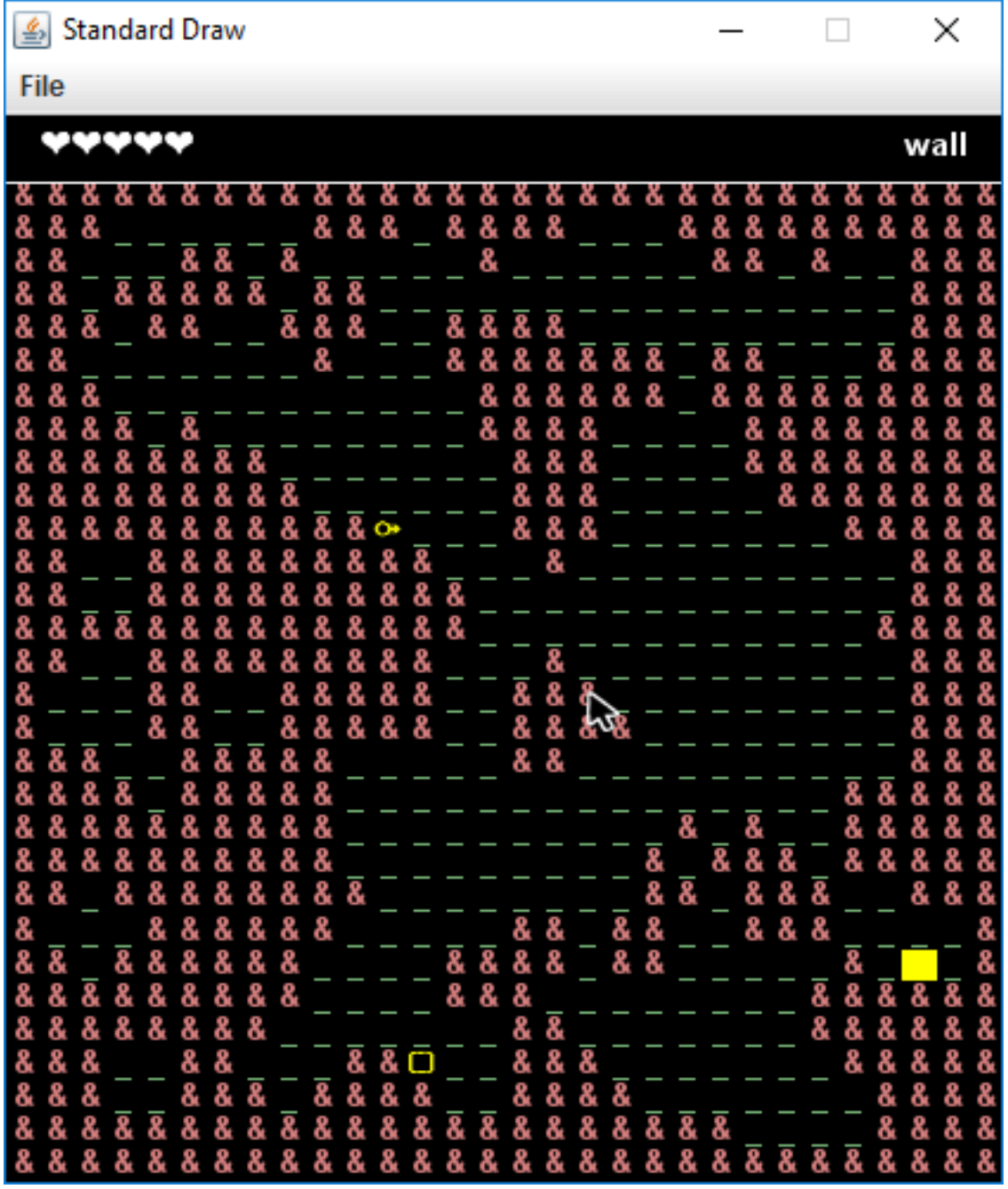
Game UI (User Interface) Appearance

After the user has entered a seed and pressed S, the game should start. The user interface of the game must include:

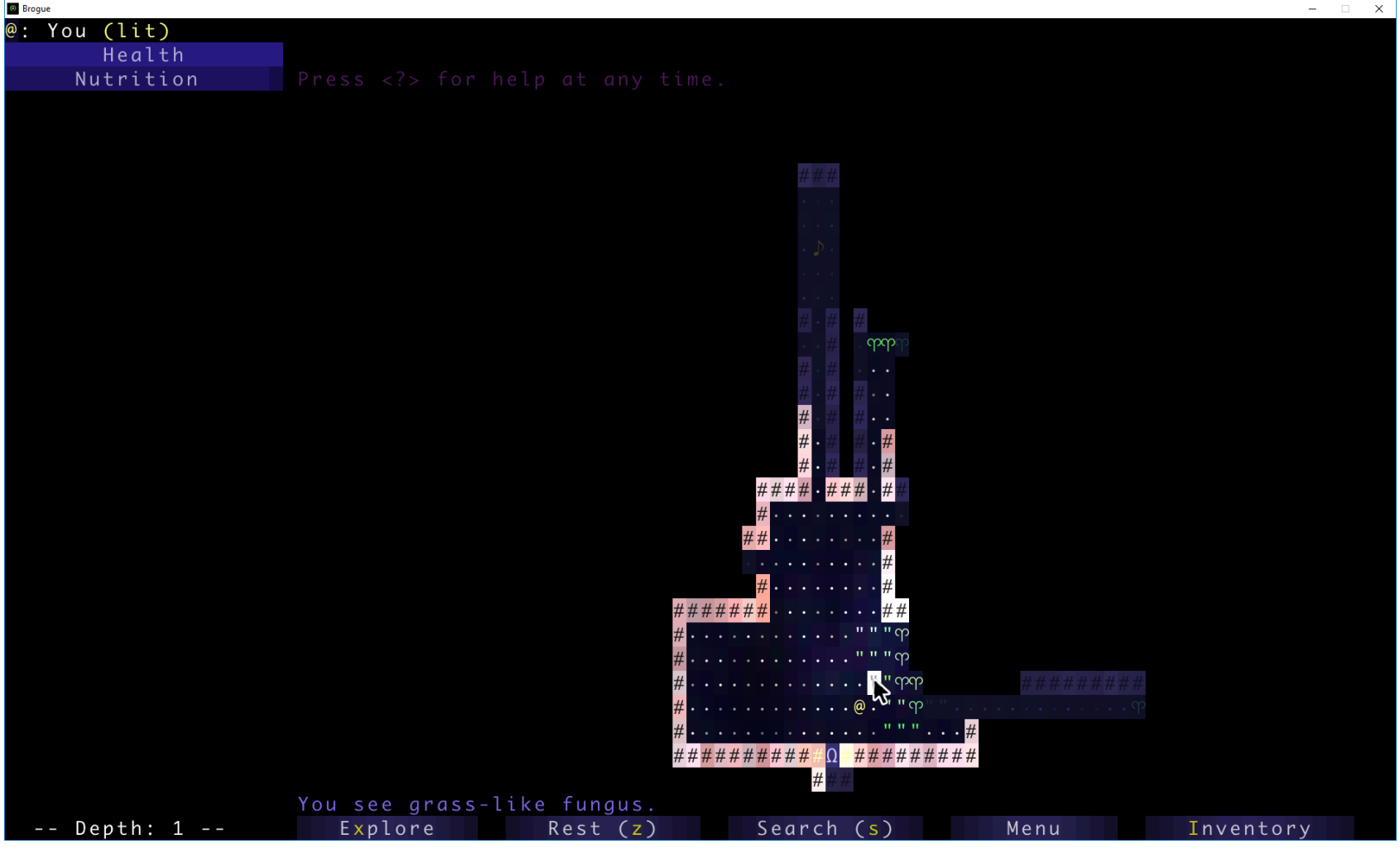
- A 2D grid of tiles showing the current state of the world.
- A "Heads Up Display" that provides additional information that maybe useful to the user. At the bare minimum, this should include Text that describes the tile currently under the mouse pointer.

As an example of the bare minimum, the game below displays a grid of tiles and a HUD that displays the description of the tile under the mouse pointer (click image for higher resolution):

You may include additional features if you choose. In the example below (click image for higher resolution), as with the previous example, the mouse cursor is currently over a wall, so the HUD displays the text “wall” in the top right. However, this game’s HUD also provides the user with 5 hearts representing the player’s “health”. Note that this game does not meet the requirements of the spec above, as it is a large erratic cavernous space, as opposed to rooms connected by hallways.



As an example, the game below (click image for higher resolution) uses the GUI to list additional valid key presses, and provides more verbose information when the user mouses-over a tile (“You see grass-like fungus.”). The image shown below is a professional game, yours isn’t expect to look anywhere near as good.



For information about how to specify the location of the HUD, see the `initialize(int width, int height, int xOffset, int yOffset)` method of `TERenderer` or see lab 6.

Game UI Behavior

When the game begins, the user must be in control of some sort of entity that is displayed in the world. The user must be able to move up, left, down, and right using the W, A, S, and D keys, respectively. These keys may also do additional things, e.g. pushing objects. You may include additional keys in your game. **The user may not interact with the world using mouse clicks**, e.g. no clicking to allow movement.

The game must behave pseudorandomly. That is, given a certain seed, the same set of key presses must yield the exact same results!

In addition to movement keys, if the user enters ":Q", the game should quit and save. The description of the saving (and loading) function is described in the next section. **This command must immediately quit and save**, and should require no further keypresses to complete, e.g. do not ask them if they are sure before quitting. We will call this single action

of quitting and saving at the same time “quit/saving”.

This project uses `StdDraw` to handle user input. This results in a couple of important limitations:

- Can only register key presses that result in a char. This means any unicode character will be fine but keys such as the arrow keys and escape will not work.
- On some computers, may not support holding down of keys without some significant modifications, i.e. you can’t hold down the e key and keep moving east. If you can figure out how to support holding down of keys in a way that is compatible with `playwithInputString`, you’re welcome to do so.

Because of the requirement that your game must be playable from a String, your game cannot make use of real time, i.e. your game cannot have any mechanic which depends on a certain amount of time passing in real life, since that would not be captured in an input string and would not lead to deterministic behavior when using that string vs. playing at the keyboard. Keeping track of the number of turns that have elapsed is a perfectly reasonable mechanic,and might be an interesting thing to include in your game, e.g. maybe the game grows steadily darker in color with each step. You’re welcome to include other key presses like allowing the player to press space bar in order to move forwards one time step.

Saving and Loading

Sometimes, you’ll be playing a game, and you suddenly notice that it’s time to go to 61B lecture. For times like those, being able to save your progress and load it later is very handy. Your game must have the ability to save the game while playing and load the game in the exact state of the most recent save after quitting and opening the game back up.

Within a running Java program, we use variables to store and load values. However, to be able to quit the game (kill the program), and then load it back up, we need to have memory that’s a bit more persistent. This

means you must save any information that is relevant to the state of the game concretely in a file somewhere you would be able to find when the game is opened again. The technique to accomplish this will be up to you, but we recommend looking into the Java interface `Serializable`, which is the easiest approach.

When the user restarts Game.main and presses L, the game should into exactly the same state as it was before the game was terminated. This state includes the state of the random number generator! More on this in the next section.

Playing With Input Strings and Phase 2

Your `Core.Game.playWithInputString(String s)` must be able to handle input strings that include movement

For example, the string "N543SWWWWAA" corresponds to the user starting a game with the seed 543, then moving up four times, then left twice. If we called `Core.Game.playWithInputString("N543SWWWWAA")`, your game would return a `TETile[][]` representing the world EXACTLY as it would be if we'd used `playWithKeyboard` and typed these keys in manually. Since the game must be deterministic given a seed and a string of inputs, this will allow players to replay exactly what happened in their game for a given sequence of inputs. This will also be handy for testing out your code, as well as for our autograder.

`Core.Game.playWithInputString(String s)` must also be able to handle saving and loading in a replay string, e.g. "N25SDDWD:Q" would correspond to starting a new game with seed 25, then moving right, right, up, right, then quit/saving. The method would then return the 2D `TETile[][]` array at the time of save. If we then started the game with the replay string "LDDDD", we'd reload the game we just saved, move right four times, then return the 2D `TETile[][]` array after the fourth move.

Your game should not change in any way between saves, i.e. the same exact `TETile[][]` should be returned by the last call to

`playWithInputString` for all of the following scenarios:

- `playWithInputString(N999SDDDWWWDDD)`
- `playWithInputString(N999SDDD:Q)`, then `playWithInputString(LWWWDDD)`
- `playWithInputString(N999SDDD:Q)`, then `playWithInputString(LWWW:Q)`, then `playWithInputString(LDDD:Q)`
- `playWithInputString(N999SDDD:Q)`, then `playWithInputString(L:Q)`, then `playWithInputString(L:Q)` then `playWithInputString(LWWWDDD)`

we then called `playWithInputString` with input "L:Q", we'd expect the exact same world state to be saved and returned as `TETile[][]` as with the previous call where we provided "LDDDD".

You do not need to worry about replay strings that contain multiple saves, i.e. "N5SDD:QD:QDD:Q" is not considered a valid replay string, since the game should have terminated before the second :Q. You do not need to worry about invalid replay strings, i.e. you can assume that every replay string provided by the autograder starts with either "N#S" or "L", where # represents the user entered seed.

The return value of the `playWithInputString` method should not depend on whether the input string ends with `:Q` or not. The only difference is whether your game saves or not as a side effect of the method.

Ambition Score

20 points of your project score will be based on features of your choosing, which we call your "ambition score". The big idea is that beyond the base requirements of this project, we want you to try and polish your product a bit more and add some cool features. Below is a list of features worth a total of 200 points. This "ambition" category is only worth 20 points, i.e. if you do 30 points worth, you do not get extra credit. However, feel free to add as many feature as you'd like if you have

the time and inclination.

Your game must still meet the basic requirements described above! For example, if you allow the user to use mouse clicks, the game should still allow keyboard based movement!

- (20 pts) Completion of 3 creative mechanics as described gold points requirements. Essentially, fulfilling the gold point requirements will also satisfy the ambition category
- (20 pts) Create a system so that the game only displays tiles on the screen that are within the line of sight of the player
- (20 pts) Add the ability for light sources to affect how the world is rendered, with at least one light source that can be turned on and off somehow
- (15 pts) Add ability to rotate the world
- (15 pts) Add entities which wander the world and destroy tiles
- (15 pts) Add turn based combat encounters which is triggered by interacting with NPCs.
- (15 pts) Modify your game so that the world map consists of more than 1 screen. This is commonly done by either having stairs which lead to a different floor and a different layout, or by having a scrolling world map which moves as the player moves to the edge of the screen
- (15 pts) Add support for 2 player gameplay/movement. Should have two playable characters on screen which can move around and have separate control schemes
- (15 pts) Add support for movement with mouse clicks on any visible square. You'll need to implement some sort of algorithm for pathfinding
- (15 pts) Add support for undoing a movement in regular play. Undoing a movement should reset the gameworld back to before the most recent keypress but should actually add to the replay string and not remove a character (ie. undo command should be logged in the replay string)
- (15 pts) Add NPCs (Non-Player Charcters) which interact with the

player. For example, adding 4 ghosts which chase the player and end the game if they are caught. For this item, there should be at least for NPCs in the game for any seed.

- (10 pts) Add a minimap somewhere which shows the entire map and the current player location. This feature will get 10 points regardless, but is a lot more interesting if you also implement a map which is larger than the screen so that you are unable to see the entire map normally.
- (10 pts) Add portals to your world which teleport the player.
- (10 pts) Add multiple save slots which also adds a new menu option and a new shortcut during gameplay to save to slots other than slot 1. You should be careful to still support the default behavior of saving and loading in order to be consistent with the replay string requirements
- (10 pts) Add some kind of leaderboard or high score list to your game. Requires there to be an objective to your game and some notion of score. Should allow you to choose a name to be entered and should display at least the top ten scores with the associated names
- (10 pts) Add animations to your gameplay.
- (10 pts) Add the ability to start a new game without closing an reopening the game, either during gameplay or when you reach a "game over" state
- (10 pts) Add an inventory to your game to store items which the player can use or equip
- (10 pts) Add a health mechanic to your game to make it more interactive
- (5 pts) Change walls every time they are touched.
- (5 pts) Add menu option to change player appearance
- (5 pts) Spiked walls which decreases a player's health
- (5 pts) "Coins" which contribute directly to
- (5 pts) Add support for mouse clicks on the main menu for anything that can be done with a keypress on the main menu
- (5 pts) Make your game render using images instead of unicode

characters
- (5 pts) Add cool music to your menu and gameplay. Also add sound effects for any interactions in your game
- (5 pts) Add some lore which is accessible from menu describing the background for your game and the story in your game
- (5 pts) Add flavor text for each tile which is also displayed next to the tile's name when a mouse hovers over it. Flavor text is just a brief description which adds more of an explanation to each tile or makes a funny joke/reference
- (5 pts) Add a display of real date and time in the Heads Up Display
- (5 pts) Add some neat easter eggs or cheat codes to your game which do something fun. Maybe add the Konami Code
- (5 pts) Add a menu option to change all text in the game to a different language. English should be the default and there should be a way to switch it back to English.
- (5 pts) Add a menu option to give your character a name with is displayed on the HUD when playing.
- (5 pts) Add a menu option or randomly determine what the environment/theme of the game will be.

This list is by no means comphrehensive of all the things you could do to satisfy Ambition points! If you have another idea for how you want to make your game really cool, fill out this form to submit your idea and how many points you think it should be worth. You will get confirmation if your idea is approved and it may be added to the Ambition list above as well. If you have multiple ideas, please fill out the form once per addition. We will link a list of approved ideas below this line as we approve them. You're welcome to use these approved ideas as well.

Requirements Summary

A list of the requirements and restrictions on your game. Please note that this section does not substitute reading the entire spec since there are a lot of details which are not captured here.

- When played with `playWithKeyboard`, must have a menu screen that

has New game, Load, and Quit options, navigable by keyboard.
- When entering New game, user should enter an integer seed followed by S key. Upon pressing S, game should begin.
- Must have pseudorandomly generated worlds/variety in games, i.e. the world should be different for every seed.
- Generated world must include all of the visual features described in phase 1 above.
- Player must be able to move around in the world using W, A, S, and D keys.
- Player must be able to press ":Q" to quit, and after starting the game up again, the L option on the main menu should load the game.
- All random events in game should be pseudorandom, i.e. gives deterministic behavior given a seed.
- Must be playable using `playWithInputString`, and behavior other than accepting input and drawing to the screen should be identical to `playWithKeyboard`.
- `playWithInputString` must return a `TETile[][]` array of the world at the time after the last character in the string is processed.
- `playWithInputString` must be able to handle saving and loading, just like `playWithKeyboard`.
- Must use our TileEngine and StdDraw for displaying graphics.
- Must have a HUD which displays relevant information somewhere outside of the area displaying the game world/tiles.
- HUD must display description of tile upon hovering over the tile.
- Must not use real life time in game mechanics.
- Features that make up 20 points from the Ambition category

Gold Points

For gold points, you should make it possible to win or lose your game, and along the way, you should also introduce 3 "creative mechanics". Mechanics refer to the underlying ways the game is controlled and how outcomes are calculated. The mechanics of a game make up the rulebook for the game and determine what can and cannot happen.

Interesting games often have interesting mechanics and interactions that lead to a large variety of game states. For gold points, we are requiring at least 3 “creative mechanics”.

We leave it up to you to define creative mechanics. We aren’t going to police this closely, so creative mechanics will be on the honor system.

After adding your mechanics as well as your win/loss conditions, create a public youtube showcasing your game, including its creative mechanics and win/loss conditions, and submit a link using [this form (link TBA)]. It is not necessary for you to able to actually win your own game, i.e. it’s OK if your game is really hard.

Grading

Autograder points: 120 points.

- Phase 1: 60 points.
- Phase 2: 60 points.
- +6 points for finishing phase 1 early.

Lab demo: 80 points.

- 20 points: Ambition of design, aesthetics, etc.
- 60 points: Obeying base spec for phase 1 and phase 2.

Gold points: 12 gold points.

Lab Demo Checkoff Script

In the hopes of keeping this process as transparent as possible, click here for the exact script the TA or tutor will use when checking your project. By the end of the demo, you should be able to determine exactly what points you received and will have an opportunity to demonstrate any feature which was not checked off during the demo but exists in your project. Note that while some of these features are a bit subjective, this demo is meant to give you a chance to defend your work and was chosen

over having us grade it locked behind doors with no input from you. We will respect the amount of work you put into your project and you should have a discussion with us if you believe we are not grading you fairly.

Office Hours

Due to the open-ended nature of this project, it will be hard for the course staff to help you debug your project in the same way that they can for other projects. As a result, we will be implementing the following procedure regarding receiving help during office hours in order to be able to allot an adequate enough time for those that attend.

- Course staff will spend at most ~10 minutes per student.
- Your code must be well documented, including all methods you write, according to the style guide. This is to minimize time spent verbally explaining what each method does.
- If your question is for debugging help, you must be prepared to explain the error that is being caused and have a test or input that can easily reproduce the bug for ease of debugging. If you come to us saying something does not work, but have not written any tests or attempted to use the debugger, we will not help you.
- When we do provide debugging help, it may be at a high level, where we suggest ways to reorganize your code in order to make clarity and debugging easier. It is not a good use of your time or the TAs' time to try to find bugs in something that is disorganized and brittle.

FAQ

Q: I want to make a game that's about exploring the outdoors or caves or something like that, not a bunch of rooms. What should I do? A: That's fine, you can just use the seed to create a starter house for your character that they can freely exit.

Q: Can I make a world that supports scrolling or multiple levels (i.e.

stairs)? A: Sure. In this case, `playWitInputString` should return only the part of the world that is visible on the screen at the time that the last character in the replay string is entered.

Q: Can I add the ability for players to customize their character before starting a game? A: Yes, but you'll need to create a fourth main menu option. Your game must support exactly the API described in this spec, i.e. "N23123S" must always start a game with the seed 23123, and must not ask for any additional input from the player.

Q: I'm getting two standard draw windows instead of one. How do I avoid having two StdDraw windows? A:Make sure you're importing `edu.princeton.cs.introcs.StdDraw` instead of `import edu.princeton.cs.algs4.StdDraw`.

Q: Why is the phase 1 autograder saying "Could not initialize class edu.princeton.cs.introcs.StdDraw"? A: Somewhere in your code, your `playWithInputString` method tries to use the `StdDraw` class which is not allowed. For example if you call `TERenderer.initialize()`, you are using `StdDraw`. No StdDraw window should open when you call `playWithInputString`. We've seen some students whose code only opens a StdDraw window for some seeds, so look very carefully.

Q: The autograder is getting a `NumberFormatException` caused by `Integer.parseInt`. A: The `Random` class takes `long` as input, so the seeds we provide are too big to fit into an `int`. You need to use the `Long` class instead to parse the seed.