Instituto Tecnológico de Costa Rica

Algoritmos y Estructuras de Datos I

Proyecto #1 - Math Sockets

Andrés Vivallo - 2020023389
Cristian Montero - 2021142076
Noemí Vargas - 2021082564

II SEMESTRE 2021

Math Sockets

Andrés Vivallo
Computer Engineering Department
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
andresvivallo4@gmail.com

Noemí Vargas
Computer Engineering Department
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
mimi2002@estudiantec.cr

Cristian Montero
Computer Engineering Department
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
cmontero255@gmail.com

***Abstract*—This document is the official documentation for the game Math Sockets. It explains the architecture of the program, the algorithms, and data structures used in the game. You'll also find some issues we ran into and the solution we (the team in charge of the development of the game) created for them.**

***Index Terms*—video game, java, libgdx, object-oriented programming, data structures, algorithms, design patterns**

I. Introduction

Link to the GitHub repo: https://github.com/Vivallo04/MathSockets

The video game MathSockets was created using the Java programming language and LibGDX as the main framework for rendering and Kryonet creating and handling the socket connection. LibGDX is a free and open-source game-development application framework written in the Java programming language with some C and C++ components for performance dependent code. It allows for the development of desktop and mobile games by using the same code base.

II. MathSockets' Architecture

A. Maintaining the Integrity of the Specifications

MathSockets is a board game for two online-players. The game consists of a nxn board, whereas the size of n is defined by the developers, the size we chose is $5x5$. There are three different types of tiles:

1) Challenge Tile
2) Tunnel Tile
3) Trap Tile

Before you begin to format your paper, first write and save the content as a separate text file. Complete all content and organizational editing before formatting. Please note sections **??**–**??** below for more information on proofreading, spelling and grammar.

Keep your text and graphic files separate until after the text has been formatted and styled. Do not number text heads—LaTeX will do that for you.

B. Class Diagram

See page 3 with Class Diagram.

C. Design Patterns

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code. Here are the design patterns we used in the creation of the game.

- Singleton: Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance. We use singleton for the Server and Board instances so that we make sure there's only one instance of each object at runtime. For instance:

```
private GServer() throws IOException {
    init();
}

public static GServer getGServerInstance(){
    if (gameSever == null) {
        gameSever = new GServer();
    }
    return gameSever;
}
```

- Observer: Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. The Observer pattern handles the server requests and sends a JSON file to the clients stored in an array.
- Abstract Factory: Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes. We use the Abstract Factory method for randomly populating the tiles on the game Board.
- State: State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class. It lets the game behave differently, depending on the State. The MathSockets class contains a reference to a StateMachine object which performs most of the work of the game.

D. Main Game States

- LoadingState: Shows the Math Sockets logo while the game is loading.
- MainMenuState: Is the state where the player can change their avatar, change the settings and decide when to start the game.
- GameState: The MathSockets game, with a Board and some player information. The players can see the current status of the game and it will be updated when a change is made by sending a request using a JSON file.
- ChallengeState: Whenever the players get over a ChallengeTile, an action is triggered and they are led to a ChallengeState where they must solve a simple mathematic challenge.
- WinState: The first player that reaches the last tile (Win Tile) in the game board will be considered the winner, and so their name will be shown on the screen.

E. Data Structures

The data structures used are the follwing:

- Doubly Linked List: A DLL (doubly linked list) is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields and one data field. We are using the Java.util implementation of LinkedList which is also a DoublyLinked list so we can meet the requirements of the project. The DLL is used for traversing the tiles on the game board, because the players move forward or backwards depending on the tiles they endup on.
- Hash Map: It's a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash map uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored. The EventManager class stores the listeners in an hash map. Whenever it's required, the corresponding clients are sent a request with the new information they have to update.

F. Algorithms

The algorithms we development the MathSockets game are the following:

```
public void render(SpriteBatch batch) {
    int currentNode = 0;
    int i = 1; // x pos
    int j = 1; // y pos
    for (Tile tile: boardNodes) {
        // in-line
        if (boardNodes.size() % i == 0 && i != 1) {
            batch.draw(tile.getTileTexture(),
            (i * tile.getWidth()) - 80,
            (Gdx.graphics.getHeight()
            - (j * tile.getHeight()))
            - tile.getHeight() / 2,
            tile.getWidth(), tile.getHeight());
            currentNode++;
            j++;
            i = 1;
        } else {
            batch.draw(tile.getTileTexture(),
            (i * tile.getHeight()) - 80,
            (Gdx.graphics.getHeight()
            - (j * tile.getHeight()))
            - tile.getHeight() / 2,
            tile.getWidth(), tile.getHeight());
            currentNode++;
            i++;
        }
    }

}
```

Depending on the entered board size the algorithm will iterate over the nodes of the DLL, get their texture and this will be drawn onto the screen, once the condition boardNodes.size() % i == 0 i != 1 is met, the position of X and Y of the tile will change to the next row.

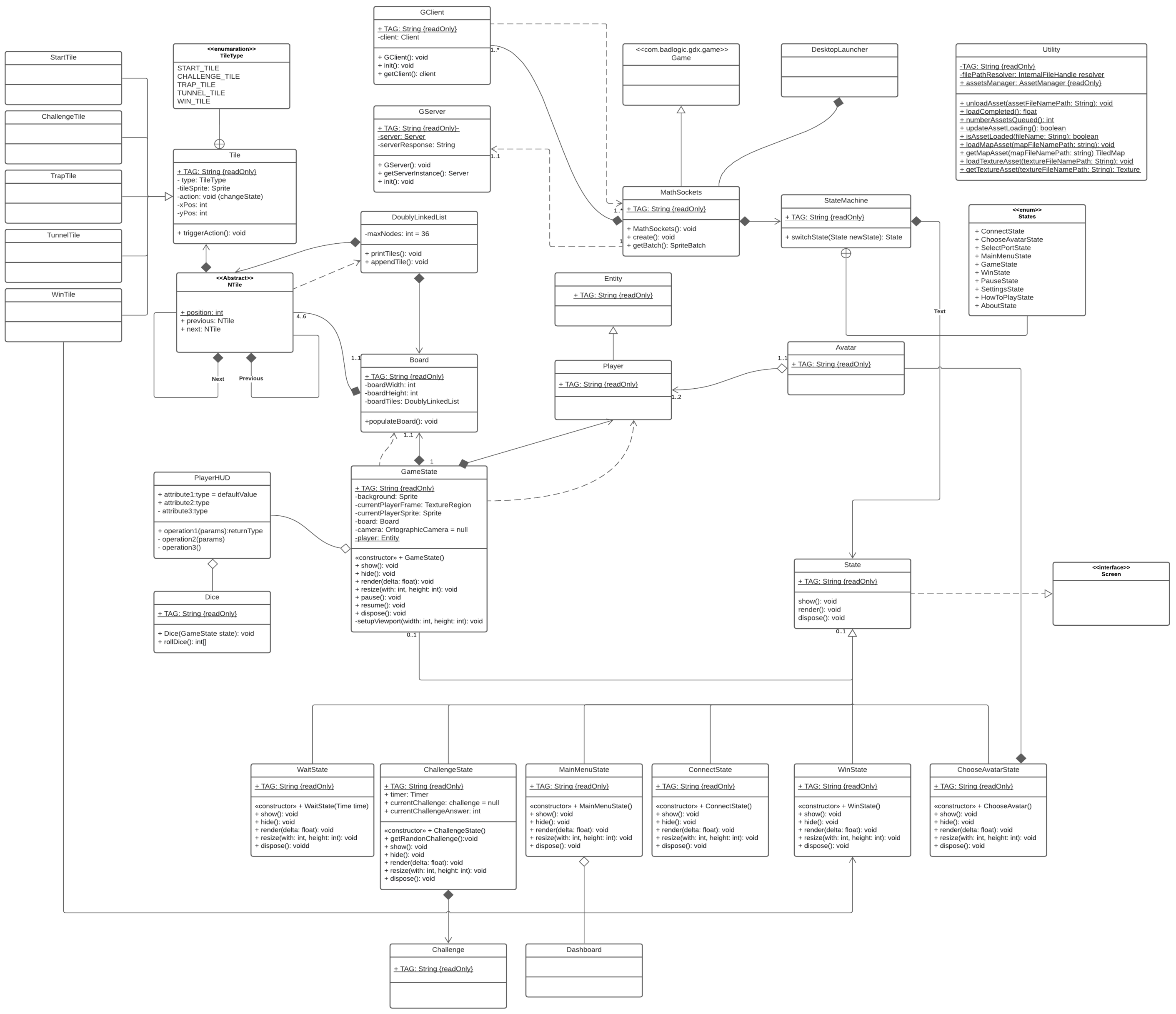
G. Bugs

We have not encountered any bugs so far.

References

[1] "The catalog of design patterns," Refactoring.Guru. [Online]. Available: https://refactoring.guru/design-patterns/catalog. [Accessed: 24-Sep-2021].

MathSockets


GClient
+ TAG: String {readOnly}
-client: Client
+ GClient(): void
+ init(): void
+ getClient(): client
GServer
+ TAG: String {readOnly}-
-server: Server
-serverResponse: String
+ GServer(): void
+ getServerInstance(): Server
+ init(): void
<<com.badlogic.gdx.game>>
Game
DesktopLauncher
Utility
-TAG: String {readOnly}
-filePathResolver: InternalFileHandle resolver
+ assetsManager: AssetManager {readOnly}
+ unloadAsset(assetFileNamePath: String): void
+ loadCompleted(): float
+ numberAssetsQueued(): int
+ updateAssetLoading(): boolean
+ isAssetLoaded(fileName: String): boolean
+ loadMapAsset(mapFileNamePath: string): void
+ getMapAsset(mapFileNamePath: string) TiledMap
+ loadTextureAsset(textureFileNamePath: String): void
+ getTextureAsset(textureFileNamePath: String): Texture
StartTile
ChallengeTile
TrapTile
TunnelTile
WinTile
<<enumaration>>
TileType
START_TILE
CHALLENGE_TILE
TRAP_TILE
TUNNEL_TILE
WIN_TILE
Tile
+ TAG: String {readOnly}
- type: TileType
-tileSprite: Sprite
-action: void (changeState)
-xPos: int
-yPos: int
+ triggerAction(): void
MathSockets
+ TAG: String {readOnly}
+ MathSockets(): void
+ create(): void
+ getBatch(): SpriteBatch
StateMachine
+ TAG: String {readOnly}
+ switchState(State newState): State
<<enum>>
States
+ ConnectState
+ ChooseAvatarState
+ SelectPortState
+ MainMenuState
+ GameState
+ WinState
+ PauseState
+ SettingsState
+ HowToPlayState
+ AboutState
DoublyLinkedList
-maxNodes: int = 36
+ printTiles(): void
+ appendTile(): void
<<Abstract>>
NTile
+ position: int
+ previous: NTile
+ next: NTile
Next
Previous
Entity
+ TAG: String {readOnly}
Player
+ TAG: String {readOnly}
Avatar
+ TAG: String {readOnly}
Text
Board
+ TAG: String {readOnly}
-boardWidth: int
-boardHeight: int
-boardTiles: DoublyLinkedList
+populateBoard(): void
PlayerHUD
+ attribute1:type = defaultValue
+ attribute2:type
- attribute3:type
+ operation1(params):returnType
- operation2(params)
- operation3()
Dice
+ TAG: String {readOnly}
+ Dice(GameState state): void
+ rollDice(): int[]
GameState
+ TAG: String {readOnly}
-background: Sprite
-currentPlayerFrame: TextureRegion
-currentPlayerSprite: Sprite
-board: Board
-camera: OrtographicCamera = null
-player: Entity
«constructor» + GameState()
+ show(): void
+ hide(): void
+ render(delta: float): void
+ resize(with: int, height: int): void
+ pause(): void
+ resume(): void
+ dispose(): void
-setupViewport(width: int, height: int): void
State
+ TAG: String {readOnly}
show(): void
render(): void
dispose(): void
<<interface>>
Screen
1..*
1..*
1..1
1
4..6
1..1
1..1
1
1..1
1..2
0..1
0..1
WaitState
+ TAG: String {readOnly}
«constructor» + WaitState(Time time)
+ show(): void
+ hide(): void
+ render(delta: float): void
+ resize(with: int, height: int): void
+ dispose(): voidd
ChallengeState
+ TAG: String {readOnly}
+ timer: Timer
+ currentChallenge: challenge = null
+ currentChallengeAnswer: int
«constructor» + ChallengeState()
+ getRandonChallenge():void
+ show(): void
+ hide(): void
+ render(delta: float): void
+ resize(with: int, height: int): void
+ dispose(): void
MainMenuState
+ TAG: String {readOnly}
«constructor» + MainMenuState()
+ show(): void
+ hide(): void
+ render(delta: float): void
+ resize(with: int, height: int): void
+ dispose(): void
ConnectState
+ TAG: String {readOnly}
«constructor» + ConnectState()
+ show(): void
+ hide(): void
+ render(delta: float): void
+ resize(with: int, height: int): void
+ dispose(): void
WinState
+ TAG: String {readOnly}
«constructor» + WinState()
+ show(): void
+ hide(): void
+ render(delta: float): void
+ resize(with: int, height: int): void
+ dispose(): void
ChooseAvatarState
+ TAG: String {readOnly}
«constructor» + ChooseAvatar()
+ show(): void
+ hide(): void
+ render(delta: float): void
+ resize(with: int, height: int): void
+ dispose(): void
Challenge
+ TAG: String {readOnly}
Dashboard