CSCI201 Final Project: Testing Document
Project Number 7: Game

Team Members
Hayley Pike: hpike@usc.edu
Daniel Santoyo: dsantoyo@usc.edu
Yi(Ian) Sui: ysui@usc.edu
Ekta Gogri: egogri@usc.edu

I. Front End- Input Interaction
   A. Main Menu Screen - Unit Testing cases
      1. Click “Login” button - should set screen to Login Screen
      2. Click “New User” button - should set screen to Register Screen
      3. Click “Login as Guest” button - should set screen to Waiting/Ready Screen
      4. Click “Exit” button - should terminate the program
   B. Login Screen - Unit Testing cases
      1. Click “Login” button
         a) - should check input is not empty(front-end)
         b) Should check if username/password exist in the database(back-end)
         c) If above checks passed - set screen to Waiting/Ready Screen
         d) If above checks failed - should display error message
      2. Click “Login as Guest” Button - should set screen to Waiting/Ready Screen
      3. Click “Back” button - should set screen to Main Menu Screen
      4. Click “Exit” button - should terminate the program
   C. Register Screen - Unit Testing cases
      1. Click “Register” button -
         a) should check input is not empty(front-end)
         b) Should check is username/password is added to the database successfully(back-end)
         c)
            (1) If above checks passed - set screen to Waiting/Ready Screen
            (2) If above checks failed - should display error message
      2. Click “Back” button - should set screen to Main Menu Screen
      3. Click “Exit” button - should terminate the program
   D. Chatbox - Unit testing cases
      1. Press “ENTER” key
         a) should send a chat message and update the chatbox.
         b) should NOT make a new line in the message.

2. Press “Send” button - should send a chat message and update the chatbox.
3. Old messages should not be saved anywhere once more than three messages have been submitted.
4. Should not be able to enter SQL database commands.
5. When a new client is connected, send a “xxx has joined” message to the other client

Chatbox Unit Test code cases

```
public class ChatTest{
        private Server server;
        public ChatTest(Server server){this.server = server;}
        //Unit test1 : check if a new message can be sent to the server
        public Boolean test1(){
                Client client1 = new Client("localhost", 6789);
                client1.connect();
                client1.sendMessage(new Chatmessage("client1", "test message1"));
                Boolean check1 = server.messageVec.get(2).getUsername().equals("client1");
                Boolean check2 = server.messageVec.get(2).getMessage().equals("test message1");
                return (check1 && check2);
        }
        //Unit test2 : check if a new message can be sent to every client
        public Boolean test2(){
                Client client1 = new Client("localhost", 6789);
                Client client2 = new Client("localhost", 6789);
                client1.connect();
                client2.connect();
                client1.sendMessage(new Chatmessage("client1", "test message1"));
                Boolean check1 = client1.messageVec.get(2).getMessage().equals("test message1");
                Boolean check2 = client1.messageVec.get(2).getUsername().equals("client1");
                Boolean check3 = client2.messageVec.get(2).getMessage().equals("test message1");
                Boolean check4 = client2.messageVec.get(2).getUsername().equals("client1");
                return (check1 && check2 && check3 && check4);
        }
        //Unit test3 : check if the size of the message vector of a client maintains as 3
        //             after 10000 messages are sent
        public Boolean test3(){
                Client client1 = new Client("localhost", 6789);
                client1.connect();
                for(int i = 0 ; i < 10000; i++){
                        if(client1.messageVec.size() != 3) {
                                return false;
                        }
                        client1.sendMessage(new Chatmessage("client1", "test message"));
                }
                return true;
        }
        //Unit test3 : check if the size of the message vector on the server maintains as 3
        //             after 10000 messages are sent
        public Boolean test4(){
                Client client1 = new Client("localhost", 6789);
                client1.connect();
                for(int i = 0 ; i < 10000; i++){
                        if(server.messageVec.size() != 3) {
                                return false;
                        }
                        client1.sendMessage(new Chatmessage("client1", "test message"));
                }
                return true;
        }
}
```

E. Waiting/Ready Screen
   1. Should display the usernames of the players
   2. Ready buttons should appear only after both players are in the waiting room
   3. Pressing the ready button sends a verification to the server.

F. Game
   1. Puzzle Pieces
      a) User can drag and drop puzzle pieces.
      b) User can NOT drag and drop puzzle pieces off of game viewport.
      c) If puzzle piece is hidden by the viewport, UI, or another puzzle piece, user can still click some part of it and drag/drop it.
      d) If puzzle pieces overlap, user grabs the top one.
      e) If puzzle piece is in its correct spot, player should NOT be able to drag and drop it.

G. Level Transition Screen
   1. User can press quit to exit the program (which then triggers a Network Connection Lost screen for the other player) (score still recorded)

II. Network Capability

A. Able to connect - If the server is running and on the same network as the client, the server should be able to accept the connection from the client and the server console should display confirmation message

B. Not able to connect - If the server is not running or on the same network as the client, an error message should be displayed on the screen. The program should not freeze.

C. Server should be able to keep tracking the size of connected serverthreads. When is connection is lost, the size of serverthreads should decrease by 1.

D. After the players enter the game, screen should display “Connection Lost” message whenever the size of serverthreads is less than 2.

III. Gameplay- backend

A. Dividing the image into puzzle pieces:
   1. Test Case 1: Checks if the program correctly creates pieces for images of different dimensions.

Sample Code: For creating puzzle pieces

```
public TextureRegion[][] sprites;
 public Texture texture;
int numOfTiles_Horizontal = 5;
int numOfTiles_Vertical = 3;
int ImageWidth;
int ImageHeight;
int PieceWidth;
int PieceHeight;

 @Override
   public void create() {
      batch = new SpriteBatch();
                        texture = new Texture("img/PUPPIES.jpeg");
                        float tile_width = texture.getWidth() ;
                        float tile_height = texture.getHeight() ;

                         ImageWidth=texture.getWidth() ;
                         ImageHeight=texture.getHeight() ;
                         PieceWidth=ImageWidth/ numOfTiles_Horizontal;
                        PieceHeight=ImageHeight/numOfTiles_Vertical;
                         sprites = TextureRegion.split(texture, PieceWidth, PieceHeight);
```

Sample Test Code: To check if the puzzle pieces are correctly created,we display all the pieces in order (to confirm the dimensions, size of pieces etc.)

```
   @Override
   public void render()
 {
            batch.begin();

            for(int row= numOfTiles_Vertical-1;row>=0; row--)
            {
              for(int col=numOfTiles_Horizontal-1;col>=0; col--)
               {
               batch.draw(sprites[row][col], 75*(col+1),75*(row+1));

//if pieces were correctly created, all the pieces would be displayed in order with the correct
spacing)
               }
             }
      batch.end();
   }
```

2. Test Case 2: Checks if program correctly creates pieces for images depending upon different game levels.

B. Scrambling the puzzle pieces
   1. Test Case 1: Checks if pieces are correctly divided between 2 players before scrambling them (to maintain good ratio of their pieces to the other player's pieces on their side).
   2. Test Case 2: Checks if pieces are scrambled while also saving their original positions.Displays the "generated" puzzle for both players and

checks if all the pieces are included & scrambled. Since this is front end, checking the resulting displayed puzzle is the best way to error check.

IV. Gameplay- front end
   A. Dragging and Dropping the Puzzle Pieces
      1. Test Case 1: Checks if dragging the piece to its original spot fixes and highlights its position.We check 2 cases: i) Dropping right piece to wrong spot, should do nothing: we should still be able to move the piece around. ii) Dropping the right piece its right spot should update the counter and fix the piece's position.
      2. Test Case 2: Checks if the users score is updated with every correctly spaced puzzle piece.We check by comparing internal counter variable with number displayed on screen.
   B. Once all pieces are in place, sends verification to server
      1. Test Case 1: If all pieces are in the right position, check if server gets the correct update.
      2. Test Case 2: The user should not be able to move any pieces and (maybe wait for the other player to finish their share of the puzzle) before being directed the the game over screen.
   C. Once both players' boards are done, logs score and transitions to level screen
      1. Test Case 1: Checks if both players see the same score
      2. Test Case 2: Checks if players are redirected to level screen only if BOTH players finished their half of the puzzle.

V. Database (JDBC) and Server
   A. White Box Test Cases
      1. Test Case 1 - The validation of a user is tested for the login functionality in the database by checking for duplicates of the username.
      2. Test Case 2 - The authentication of the login functionality for a user is tested by checking if the account (e.g. userID, username, and password) exists in the database. An error message would be sent to the front end via the server to display to the user: "Invalid login."
      3. Test Case 3 - User login functionality is tested for the matching of a username and password in the database. An error message would then be sent to the front-end through the server to disclose to the user that the username and password do not match.
   B. Black Box Test Cases
      1. Test Case 1 - The user attempts to sign in with a username and password that do not match.
      2. Test Case 2 - The user can create multiple accounts with the same username?
   C. Unit Testing Case
      1. Unit testing code to test if the database would store duplicates of an account:
   D. Stress Testing Cases

1. Test Case 1 - The amount of storage of accounts in the database is tested.

Unit Test Cases Code:

Database (JDBC)

```
package test;

import java.sql.ResultSet;
import java.sql.Statement;

import project.server.JDBCType;
import project.server.Username

public class DatabaseTest {
	@Test

	private String usernameOne = "Player1";
	private String userNameTwo;
	ResultSet rs = null;
	Statement st = null;
	boolean isMoreThanOneRow;
	userNameTwo = nameTwo.getUsername();
	isMoreThanOneRow = false;

	public boolean testDatabase() {
		JDBCType userTest(usernameOne, usernameTwo);

		rs = st.executeQuery("SELECT username FROM user_table");
		isMoreThanOneRow = rs.first() && rs.next();
		return isMoreThanOneRow;
	}
}
```

CSCI201 Final Project: Detailed Design Document

Project Number 7: Game

Team Members
Hayley Pike: hpike@usc.edu
Daniel Santoyo: dsantoyo@usc.edu
Yi(Ian) Sui: ysui@usc.edu
Ekta Gogri: egogri@usc.edu

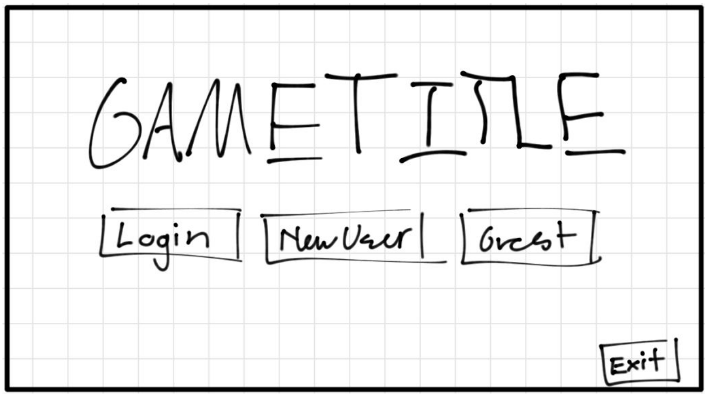
Hardware/software requirements

- User
    - Hardware:
        - RAM: 512 MB or more
        - Disk space: 512 MB or more
        - Processor: Minimum Pentium 2 266 MHz processor
        - Keyboard
        - Monitor with minimum 1024 * 768 resolution
        - Internet connection
    - Software:
        - Operating System: 64-bit
            - Windows: Windows 7 or newer
            - MacOS: 10.9 Mavericks or newer
        - A Java 8 or newer JRE/JDK
- Programmer
    - Hardware:
        - RAM: 512 MB or more
        - Disk space: 512 MB or more
        - Processor: Minimum Pentium 2 266 MHz processor
        - Keyboard
        - Monitor with minimum 1024 * 768 resolution
        - Internet connection
    - Software
        - Operating System: 64-bit
            - Windows: Windows 7 or newer
            - MacOS: 10.9 Mavericks or newer
        - A Java 8 or newer JRE/JDK

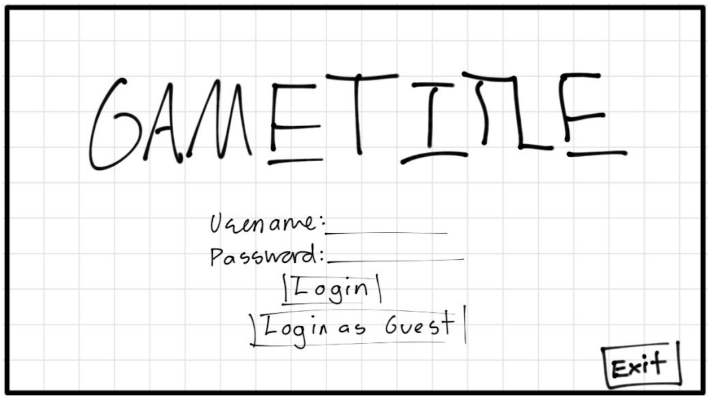
- Eclipse Photon (4.8) or newer

GUI Mockup

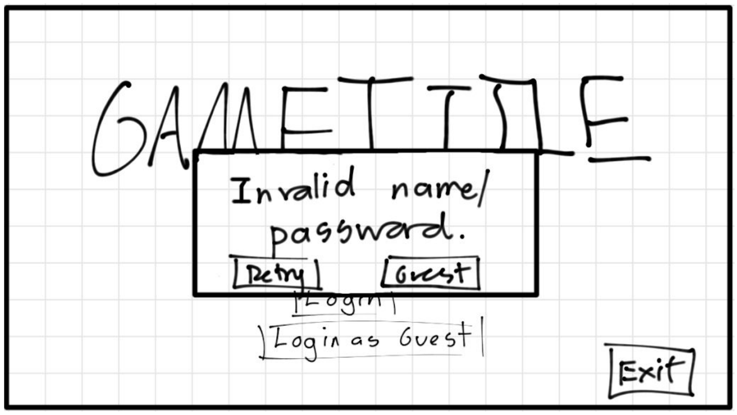
Starting Screen



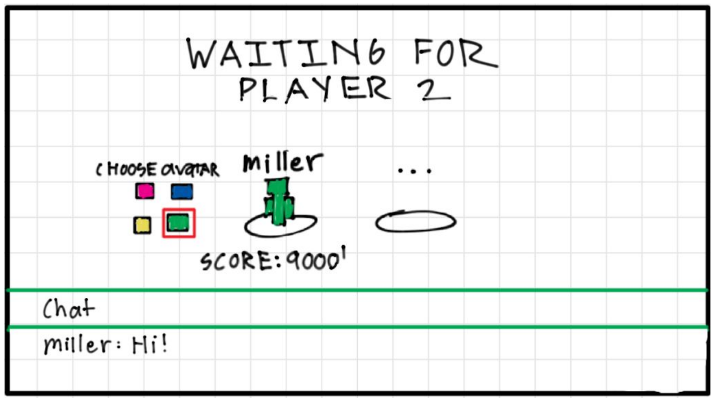
Login/Register Screen (pretty much the same thing)



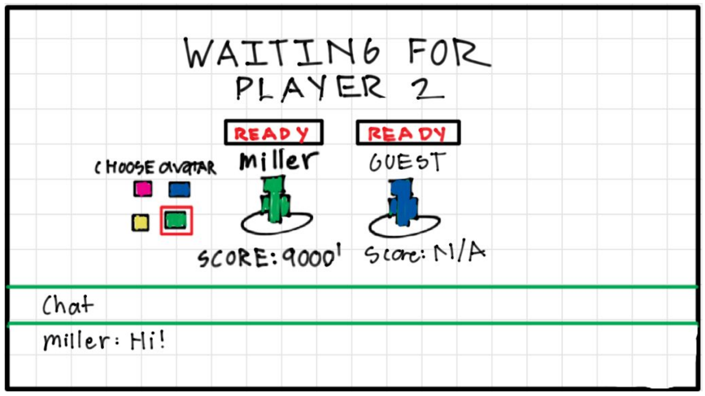
Invalid Login Screen



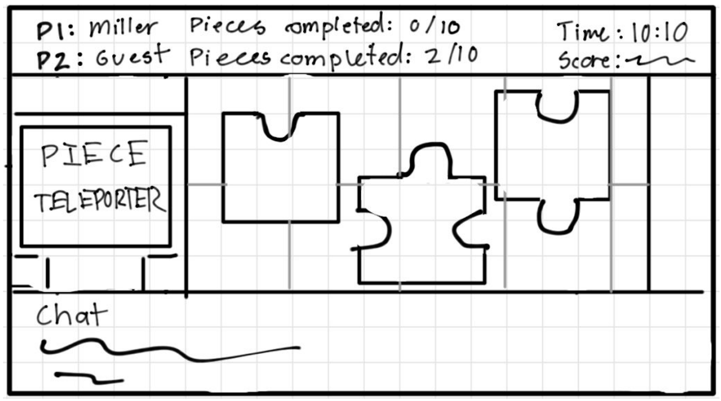
Waiting for Player Screen



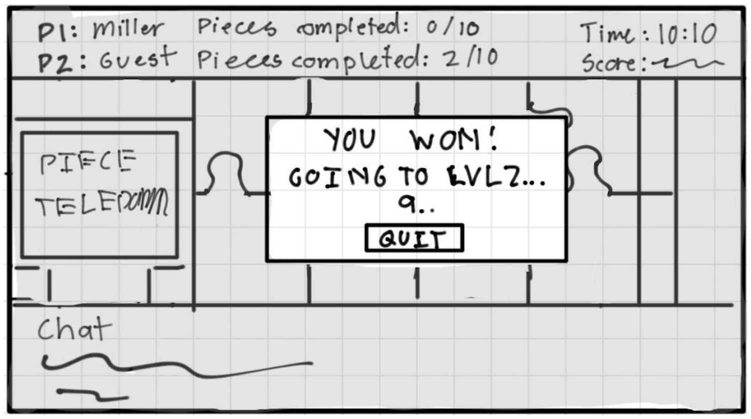
Ready to Start Screen



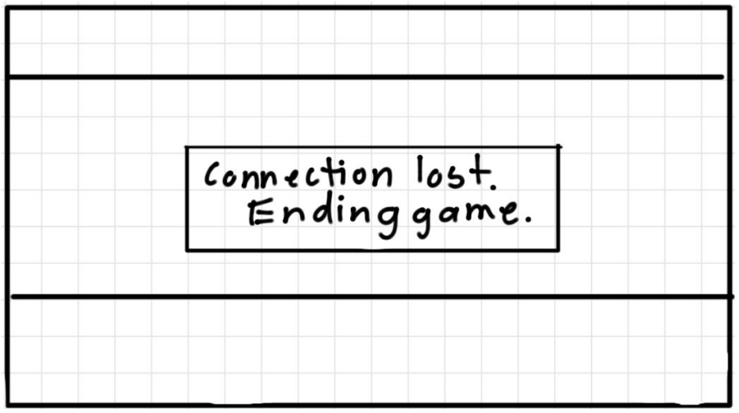
Main Game Screen (interact with mouse)



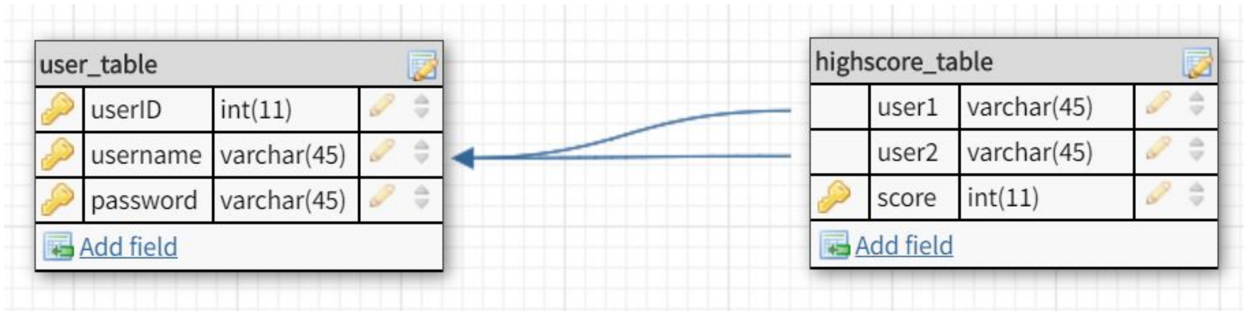
Level Transition Screen


P1: Miller Pieces completed: 0/10
P2: Guest Pieces completed: 2/10
Time: 10:10
Score:
PIECE
TELE
YOU WON!
GOING TO LVL2...
9..
QUIT
Chat

Connection Lost Screen



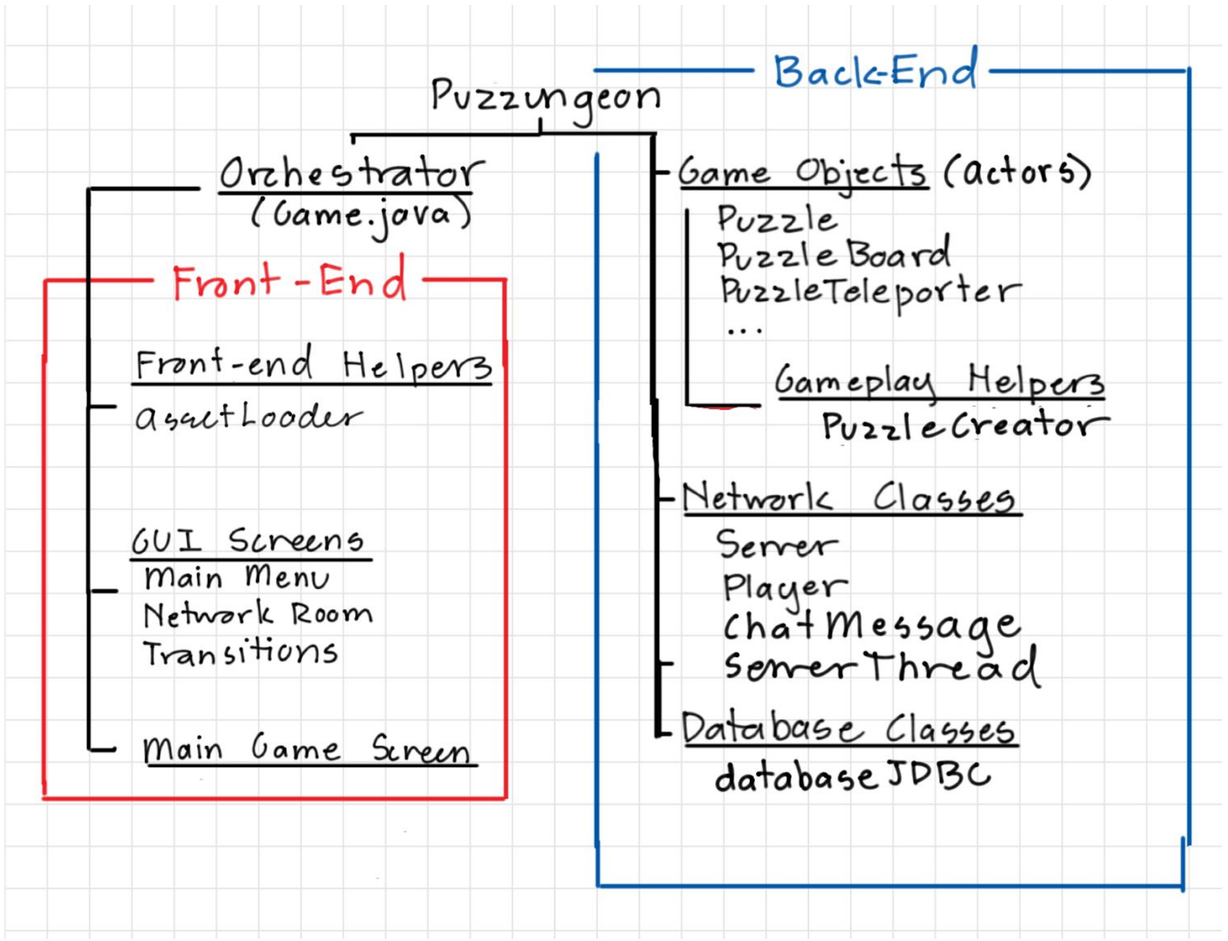
Database schema



user_table		
🔑	userID	int(11)
🔑	username	varchar(45)
🔑	password	varchar(45)
	Add field	

highscore_table		
	user1	varchar(45)
	user2	varchar(45)
🔑	score	int(11)
	Add field	

Class Diagram and Inheritance Hierarchy



Classes

Orchestrator

Class: Puzzungeon.java extends Game Extends ApplicationListener (libGDX class), the core game class. Creates game and objects, runs render function, and disposes of assets upon program termination.

private Client client // connects to the server
 	
//First screen. When new screens are needed they will be called inside the current Stage's method.
Public Stage mainMenu
 	
Public void create() //loads assets and calls first screen
 	
Public void dispose() //disposes visual assets in order to free up memory (called when switching screens and at end of program). Just a wrapper for AssetLoader's dispose().
 	
Important method: setScreen(Stage stage)
//switches active Stage
//Each Stage object(com.badlogic.gdx.scenes.scene2d) handles part of the display of the
//game(input/image)

Asset Loader

Class: AssetLoader Static class; loads text, image, and sound assets for all game objects.

Public static Skin uiSkin;
Public static TextureRegion puzzle1, puzzle2;
Public static BitmapFont titlefont, bodyfont, scorefont;
Public Sound music, click, pieceSuccess;
Public Texture background, puzzleTeleporter;
 	
Public static void load() //loads all the assets
 	
Public static void dispose() //clears all assets from memory. Called by Puzzungeon.dispose().

GUI Screens

Class: MainGameScreen implements Screen
Each Stage object(com.badlogic.gdx.scenes.scene2d) handles part of the display of the game(input/image)

Main Gameplay screen

Puzzungeon game; //reference to the game

Public MainGameScreen(puzzungeon game)
//sets game screen resolution, adds stage and its actors (game objects and UI), adds input processors, adds events listeners for UI

Public void act() //updates actors

Public void draw() //draws actors

Class: MainMenuScreen implements Screen
Each Stage object(com.badlogic.gdx.scenes.scene2d) handles part of the display of the game(input/image)

First screen; login/validation happens on this screen.

Puzzungeon game; //reference

Public MainMenuScreen() //sets game screen resolution, adds stage and its actors (game objects and UI), adds input processors, adds events listeners for UI

Public act() //updates actors

Public draw() //draws actors

Class: waitingRoom extends Screen
Each Stage object(com.badlogic.gdx.scenes.scene2d) handles part of the display of the game(input/image)

Network screen; network connection happens on this screen; server should let each client know if any other client is also connected

Puzzungeon game; //reference

Public waitingRoom() //sets game screen resolution, adds stage and its actors (game objects and UI), adds input processors, adds events listeners for UI

Public act() //updates actors

Public draw() //draws actors

Public void ifConnected()
//use a thread to continuously check if another client has connected to the server

Class: startGame extends Screen
Each Stage object(com.badlogic.gdx.scenes.scene2d) handles part of the display of the game(input/image)

Transitional screen; no input from user

Puzzungeon game; //reference

Public startGame() //sets game screen resolution, adds stage and its actors (game objects and UI), adds input processors, adds events listeners for UI

Public render() //updates and draws Actors in Stage

Class: levelTransition extends Screen
Each Stage object(com.badlogic.gdx.scenes.scene2d) handles part of the display of the game(input/image)

Transitional screen; user can choose new level or quit the game

Puzzungeon game; //reference

Public levelTransition() //sets game screen resolution, adds stage and its actors (game objects and UI), adds input processors, adds events listeners for UI

Public render() //updates and draws Actors in Stage

Class: endGame extends Screen

Each Stage object(com.badlogic.gdx.scenes.scene2d) handles part of the display of the game(input/image)

Transitional screen; appears when game is over, player can exit the game.

Puzzungeon game; //reference

Public endGame() //sets game screen resolution, adds stage and its actors (game objects and UI), adds input processors, adds events listeners for UI

Public render() //updates and draws Actors in Stage

Class: networkError extends Screen
Each Stage object(com.badlogic.gdx.scenes.scene2d) handles part of the display of the game(input/image)

Transitional screen; appears when network connection is lost. Player can only exit game.

Puzzungeon game; //reference

Public networkError() //sets game screen resolution, adds stage and its actors (game objects and UI), adds input processors, adds events listeners for UI

Public render() //updates and draws Actors in Stage

Public draw() //draws actors

Gameplay Objects

Class: Puzzle extends Image
Stores game object assets, input events

@Override private Vector2 position;

Private boolean inPosition;

Actors (Puzzle extends the libgdx Image class, which is an Actor that can be moved and placed in position)

public void displayPieces(): displays each scrambled piece as an individual actor

public void movePiece(): if not in correct position, move the pieces by implementing the “drag and drop” class
public bool checkPiece(): once an actor is moved, check if it is within bounds of its correct place on the puzzle board or the puzzle teleporter. If in correct place, lock in place. If within teleporter bounds, call sendPiece(). If neither, do nothing.
public boolean checkPuzzle(): once all pieces are in place, we wait for the other player or redirect to the game over screen
public void sendPiece(): if in bounds of the Teleporter, send piece(actor) to the stage of the other player

Class: PuzzleBoard extends Image
Stores game object assets, input events

@Override private Vector2 position;

Actors (PuzzleTeleporter extends the libgdx Image class, which is an Actor that can be moved and placed in position)

//(no direct input handling; handled in Puzzle class)

Class: PuzzleTeleporter extends Image
Stores game object assets, input events

@Override private Vector2 position;

Actors (PuzzleTeleporter extends the libgdx Image class, which is an Actor that can be moved and placed in position)

//(no direct input handling; handled in Puzzle class)

Server Classes

Class: chatMessage
each instance stores a message and the username of the sender

public static final long serialVersionUID
private String username
private String message

public ChatMessage(String username, String message) public String getUsername()

public String getMessage()

Class: ServerThread (extends Thread)
When a client object connects to the server, the server assigns it to a serverthread object
This object is used to handle data transfer between the server and the client

Networking Logic:
private ObjectInputStream ois
//catch input from the client class and send to the server

private ObjectOutputStream oos
//send data from the server to the client

private Server cs
// each ServerThread is linked with a server object

public ServerThread(Socket s, Server cs) // constructor
public void run()

(chatbox logic)
public void sendMessage(ChatMessage cm)
//send chat message from the server to the client

Other game logic:

Puzzle puzzleOject
//Puzzle object used to call the sendPiece() method to send puzzle pieces between the stages of clients

Class: Server
back-end server. Should be combined with any other back-end logics

Network logic:
private Vector<ServerThread> serverThreads
// each ServerThread represents a client

public Server(int port)
// constructor. Initialize messageVec and serverThreads.
// assign new connection from a client object to a serverthread

(chatbox logic)
private Vector<ChatMessage> messageVec

// store chat message from every client

public void broadcast(ChatMessage cm)
// sending the newest message in server's messageVec to every client

Game Objects:

Game Helpers:
createPieces(): splits the image into individual pieces
scramblePieces(): scrambles the pieces randomly and divides them between both users

public void clientsToServer()
//let each client know how many clients are connected to the server. For waiting room stage

Class: Client
store in-game data and connect to the server

private String username
private String password
private int userID
private int score

public Client(String hostname, int port)
//constructor. initialize messageVec. set up connection with the server. Read client information from the server

Network related:
private Socket s;
private ObjectInputStream ois;
private ObjectOutputStream oos;
private int port;
private String hostname;

chatbox logic
private Vector<ChatMessage> messageVec
//stores chat message sent from the server and is used to can display the messages

Database Classes

Class: DatabaseJDBC Database logics: Create and store usernames/passwords

public bool loginValidation(String username, String password)
//Validates the user login by checking the matching of username and password
 	
public int getHighScore()
//returns the high score of the client.
 	
public bool ifGuestExists(String username)
//Checks if a second player is created as guest
 	
public void addSecondPlayer(int limit)
//Creates player 2 via a similar method of creating player 1
 	
Connection conn //Used to establish a connection
Statement st //Creates a statement and sends it to the database
ResultSet rs // Access certain keys from table(s) in the database
PreparedStatement ps //References a specific column of a table from the database
Scanner scan //Read Strings
 	
//The ID from the player that gets stored/accessed via the ResultSet object in the database
Private int ID
//Similar to the above: Client score to be stored/accessed via the ResultSet object
Private int score
//Similar to the above: Client username to be stored/obtained via the ResultSet object
Private int limit
//Set limit to 2 so no more than two players can be created for a session.
Private String username
//Similar to the above: Password for the specific Client username stored/obtained via the ResultSet
Private String password

CSCI201 Final Project: Technical Specifications

Project Number 7: Game

Start menu interface(3 hours)

The start menu interface has 4 buttons: “Login”, “New User”, “Guest”, “Exit”
If the user clicks “Login”, redirect the user to the login interface.
If the user clicks “New User”, redirect the user to the register interface.
If the user clicks “Guest”, redirect the user to the waiting room interface.
If the user clicks “Exit”, the program terminates.

Login interface(4 hours)

The login interface has a username field, a password field, a “Login” button, a “Guest” button” and a “Exit” button

Our program verifies the entered login information in our database. If the entered username/password combination is correct or the user chooses “Guest”, the user is redirected to the waiting room interface.

If the entered information is incorrect, a error message box will jump out.
If the Exit button is clicked, the program terminates.

Register interface(4 hours)

The register interface has a username field, a password field and then the “Save as new User” button. Then the data is saved to the database and the user is redirected to the waiting room interface.

Chatbox interface(4 hours)

A chat box interface existed at the bottom of game since players enter the waiting room interface, until a game ends. The chat box is divided into two parts:
A input box which allows the players to enter and send their message
A display region which shows the chat messages(they can only see the newest message)

Waiting room interface(8 hours)

The waiting room interface has a display of text that discloses the player to wait for the other player to log-in and join the game.
Once both players log-in:

1) They are prompted to choose their avatar(guest players can't choose their avatar)
2) Display's the chat box as the page footer (that remains constant throughout the duration of the game)
3) Once they chose the avatar, a "Confirm Avatar" button is displayed and the user's redirected to the "Ready Interface".

Ready Interface(4 hours)

Similar to the Waiting room interface but displays both players chosen avatar with their high score and who they got the high score with (if exists) and username (if exists). Then both players click the "I'm ready" button and once both players are ready the game starts and the players are redirected to the " Start Game Interface".

Start Game Interface(2 hours)

We display the storyline (10 sec and fades out) and then divulges the player(s) the controls on how to play the game (10 sec and fades out). Then there is a countdown and the game begins as the players are redirected to!

Making the puzzle game

The two players must work together to finish an image puzzle. Each player can only see half of the puzzle and must communicate to tell each other about the image, ask for pieces, etc.

A) **Dividing the image into puzzle pieces (3 hours)**
We statically save a few images and based on the difficulty level, we break it up into individual pieces.

B) **Scrambling the puzzle pieces and dividing them between the two players (3 hours)**
Once we have all the pieces, we first scramble them randomly and allocate about half to each player.

C) Managing how the users view their side of the puzzle and move the pieces (8 hours)
A player only sees half of the image and needs pieces from the other player's side to finish their half. Players are given the ability to pass their pieces to the other player and talk to each other using the chat box. They can move the pieces using mouse click navigation.

D)Checking to see if the completed puzzle in correctly in place (3 hours)

Once both players successfully complete their half of the puzzle, we show them the final image and they are directed to the gameover interface!

In-game interface(8 hours)

Bottom Bar (that remains constant throughout the duration of the game)

Display the chat box

Top Bar (that remains constant throughout the duration of the game)

Display the current score(time that has passed),

the name of the users. We also display the current progress of each player (how many pieces they've correctly placed already)

Controls

Players use their keyboard to control the movement of their avatar

- Game with avatars
    - Effectively move around a space
    - Pick up and carry around objects
    - Simultaneous action
    - Interacting with each other/each other's objects

Gameover interface(2 hours)

Once the players finish the level, we display their final score! And display if this their personal best! If both players are registered, we give them an option to play Level 2 or they can exit the game. If either or both players are guests then they can't access Level 2 and are given an option to exit the game! The gameover interface displays a message connected to the storyline of the game and a "Thanks for play!" message.

Lost player/Lost connection interface (2 hours)

The lost player or lost connection interface displays a message (e.g. text box) that a player quit or lost connection to the game. Subsequently(after a certain amounts of time), the game session ends and the player(s) are directed back to the start menu interface.

Database (4 hours)

- There are 2 tables in the database: a user table and a high score table.
- The user table will store usernames and passwords, and will be used to authenticate the user on log-in.

- The high score table will be a junction table with a score variable and two foreign keys, both pointing to the user table. To store high scores (which are associated with a pair of users as this is a co-op game), we’ll input the userID of player 1 into the first foreign key column and the userID of player 2 into the second foreign key column, and then store the score.
- High scores will usually be displayed with the names of both players.
- We will have one userID for a ‘Guest’ user, but only in order to display a partner in a registered user’s score. Guests won’t be able to see their scores in game.

CSCI201 Final Project: High-Level Requirements

Project Number 7: Game

Team Members
Hayley Pike: hpike@usc.edu
Daniel Santoyo: dsantoyo@usc.edu
Yi(Ian) Sui: ysui@usc.edu
Ekta Gogri: egogri@usc.edu

Overview

We need to create a 2D multiplayer co-op puzzle solving game. Two players are put in a room, and they need to move around and use objects they find to solve puzzles(listed below) in order to escape from the room. There should be a user profile interface to display the user's game history. Two players should be able to interact with each other in the game, and there should be a chatbox for them to exchange information. The game will ask a new player to register a user account, and the player can use this account to login in the future. If a new player choose to play as a guest, the gaming experience is limited(registered users get additional benefits). There should be a waiting room interface for the players to wait for each other to join. A time limit is implemented and linked to the score/reward system.

List of High-Level Requirements

- Game with avatars
  - Effectively move around a space
  - Pick up and carry around objects
- Co-op multiplayer
  - Simultaneous action
  - Interacting with each other/each other's objects
  - Waiting for two players to join before starting the game
- Communication between players
  - Chatbox
- Puzzle-solving
  - Hazardous puzzles
  - Time limit links to the score/reward system
- Login/register capability
  - Registered users get in-game benefits. Guest users don't.

    Add: guest can't choose their avatar. They use the default avatar

    Add: guest can't see their current score in-game
  - Score system
- User profile
  - Display game history(playing with whom/ score)