First Fantasy Wars MK.II

CSC 631 Multiplayer Game Development

Final Team Documentation

Spring 2018

San Francisco State University

Team Avalanche

Alonzo Contreras – Team Lead/Database
Richard Robinson – Client
Justin Zhu – Client
Dalu Li – Server and Integration
Ralph Acosta – Concept and Integration
Mehi Ludwon – Art and Design

Contents

Game Overview

First Fantasy Wars is a turn-based point and click multiplayer game in development that is made/played on the unity engine. It is a 2-player game where each player starts off with 5 units which can be chosen from 3 base classes; Warrior: melee class, Mage: range class, and Bard: support class. Each unit has 3 abilities that the player can choose from a pool of 5 different abilities and these can be customized on the loadout scene before the game. These abilities that are customized are saved to be later used in future games. The objective of each player is to eliminate all the opponent's units to win.

Disclaimer: In the current state of the game, multiplayer is only supported locally. Players can play with two teams against each other, but units for second player are preset. Spawning was done across the map with one player's units on one side for an iteration, but for the presentation we had moved spawning closer to be able to show abilities without walking across the whole map. The submitted version of the game has connection to a server, registration, log in, and a few other components done.

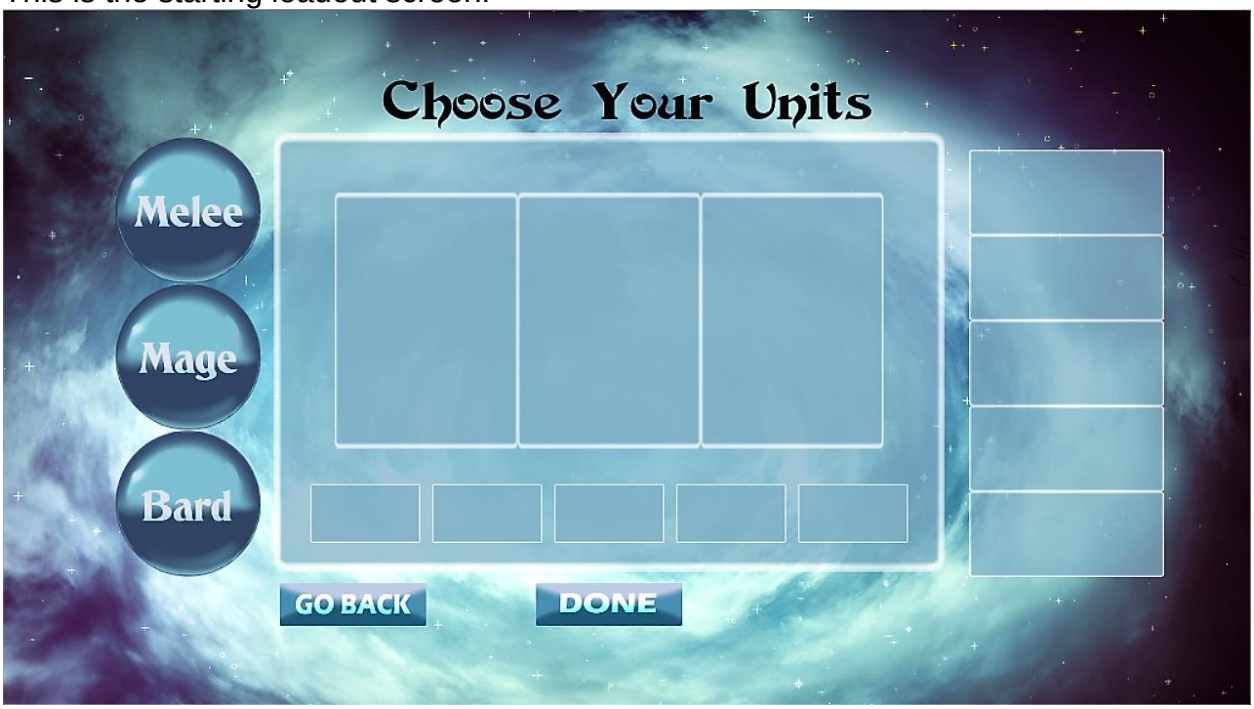
Update: We received an extension to help integration and due to some circumstances on the server code, we had to modify the client side away from the original plan in order to accommodate the unfinished server code. **Original Idea** title will be placed over the original work and **Modified** title is for the items changed to accommodate the server code.

Game Rules and How to Play

This is an example of instructions for the current state of the game.

To test the game as is please run MapOne Scene first. All other scenes depend on the server being up and running in order to avoid NULL Pointers. Once server is up and connected, you can run from TitleScene.

This is the starting loadout screen.

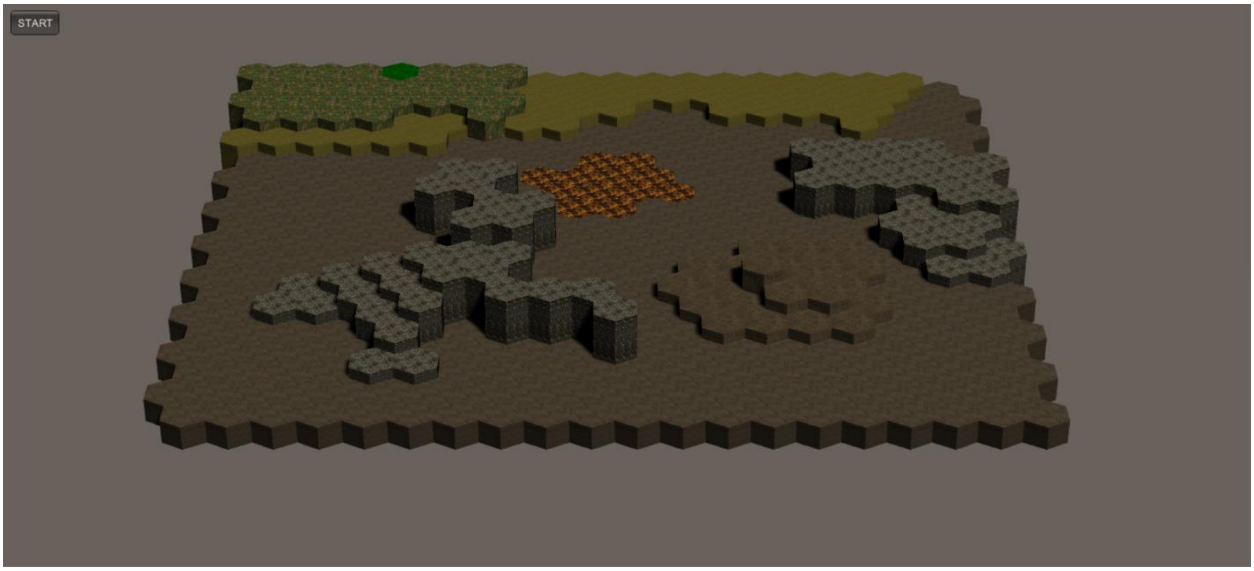


Instructions on choosing units and abilities:
1. Select a unit box
2. Select the unit you wish to have in your loadout
3. Select an ability box
4. Select the ability you wish to have for your current unit
5. Repeat for all units

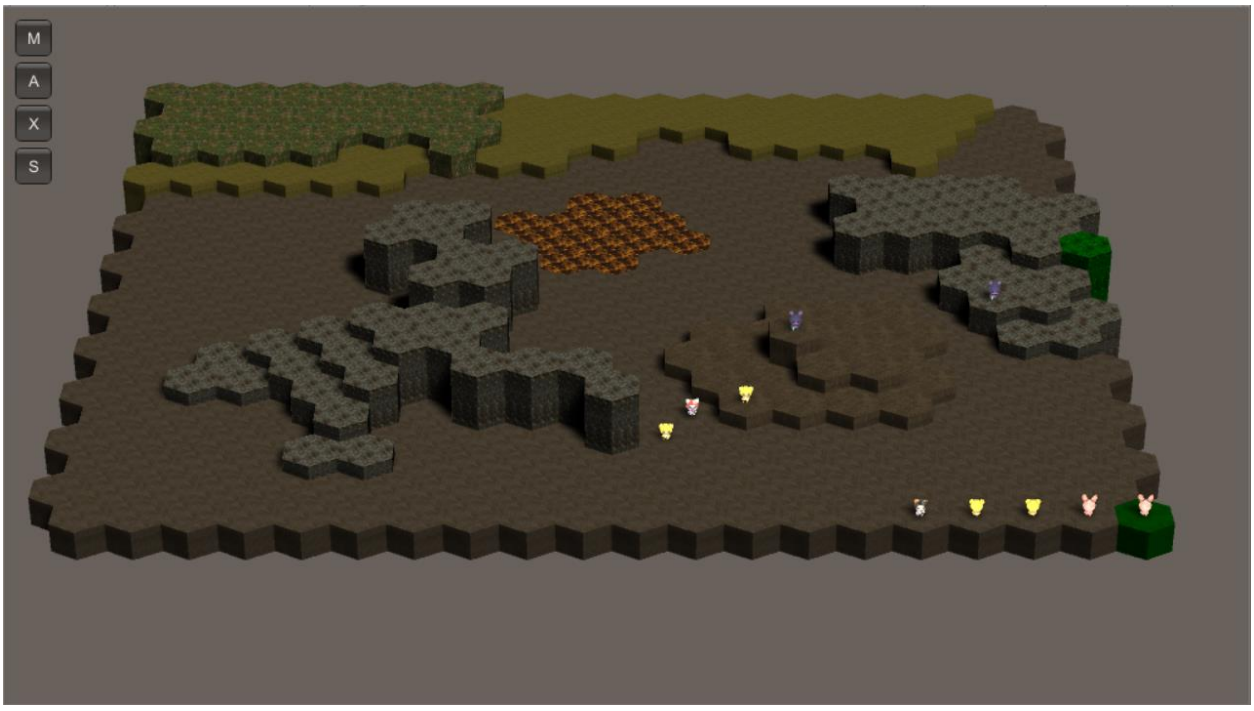
Modified:

The loadout scene has been updated/modified on the server-client integration section of this documentation.

Once you have clicked done, player will then be taken to the map



When the player clicks start, the units are then loaded onto the map

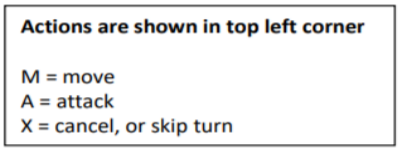


The turn sequence works as follows:

Each player takes turns:
1. Player one moves one unit
2. Player two moves one unit
3. Sequence repeats as long as players have units to control

Players do not choose which unit to use each turn specifically and instead only choose the initial unit sequence which is rotated through.

Controls: mouse-click on the buttons for the actions desired



When the ‘M’ button is clicked:
-This highlight movable tiles, mages have movement range of 2.
-This uses A*pathfinding for movement.

When the ‘A’ button is clicked:
-Ability choice is shown in top left corner.
-Select ability number to use. (Ability name are not currently shown)
-Currently the button does not display what ability is in the slot. This could be rectified by accessing the name of the ability and displaying it on the button.

When an ability number is clicked:

-Highlights attackable tiles

-Click on a tile to attack

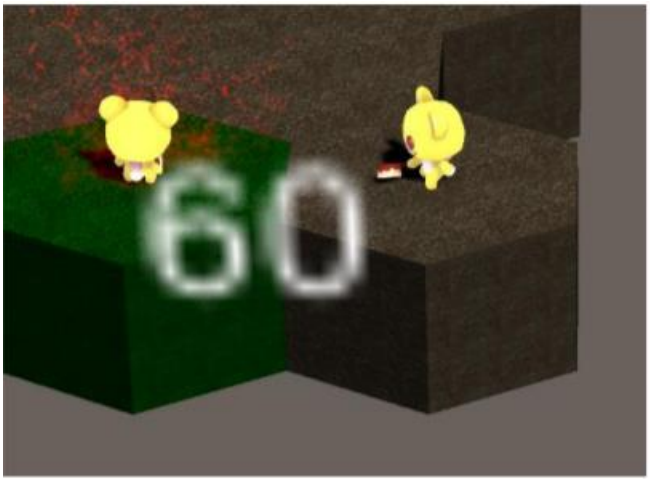
Below is the list of abilities currently implemented. They are not 100% complete in all ways, they were supposed to work, but the baseline is there with animations and voiceovers/sfx. Cooldowns are not currently used but could be implemented easily.

MELEE

Charge

-DO NOT use charge on a unit or on tiles that cannot be reached by walking, it will lock up. This is a current bug that needs to be fixed. It can only move to what would be walkable by that unit.

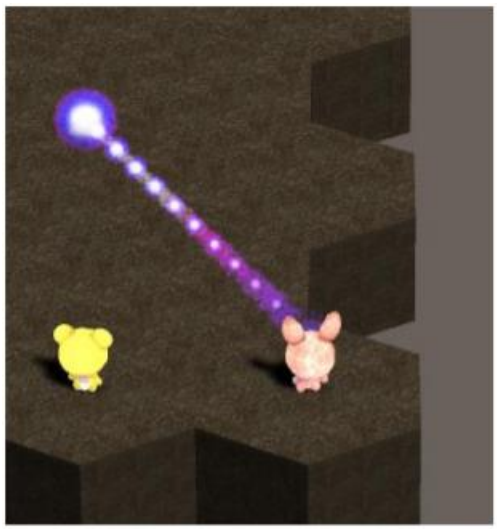
Brutal Blow



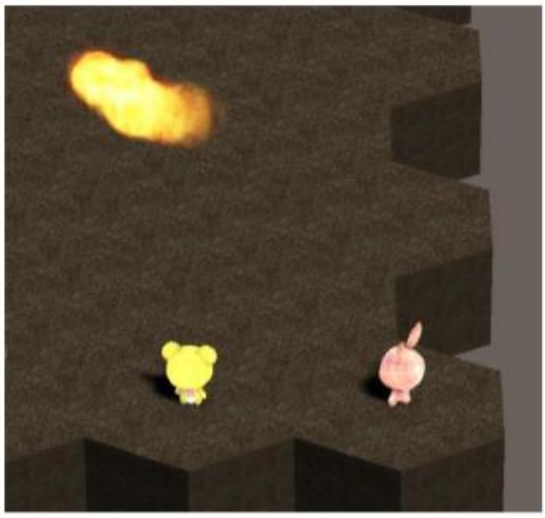
Poison Blades

MAGE

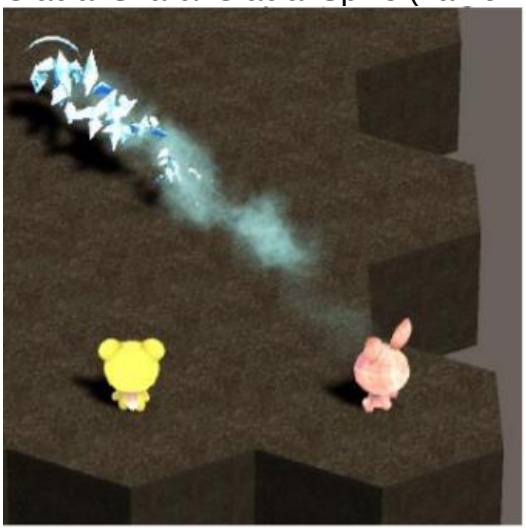
Arcane Missiles

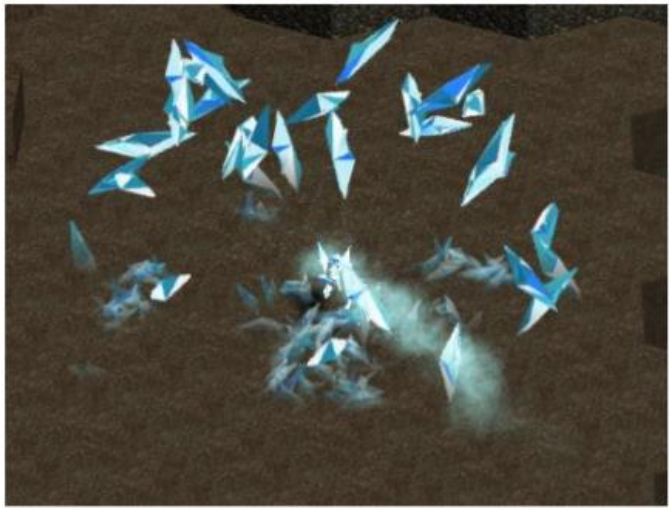


Fireball

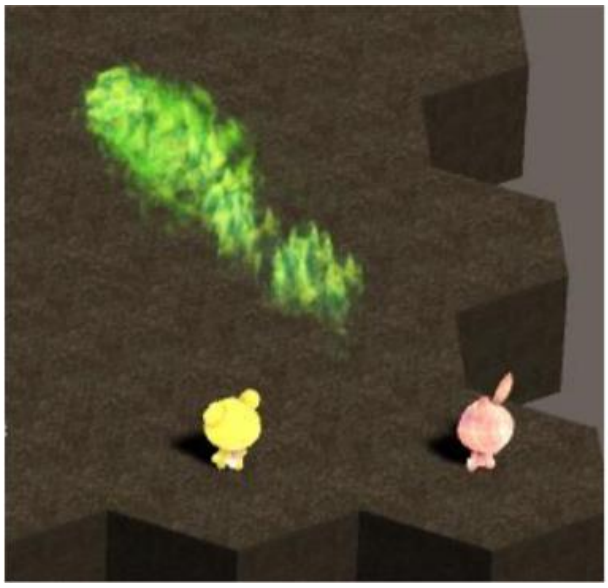


Glacial Shard/ Glacial Spike (name incongruency)





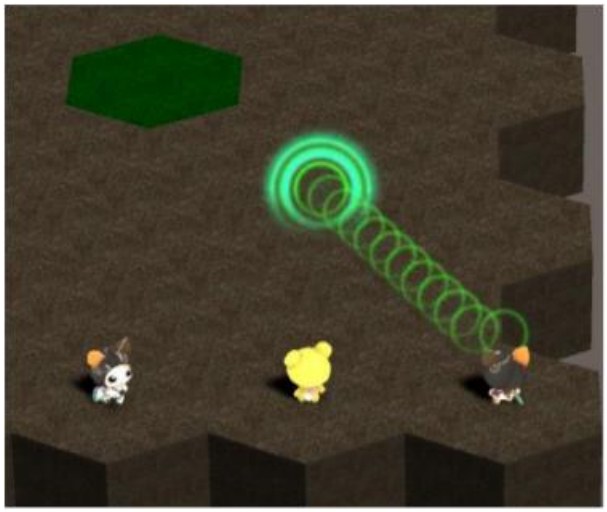
Poison

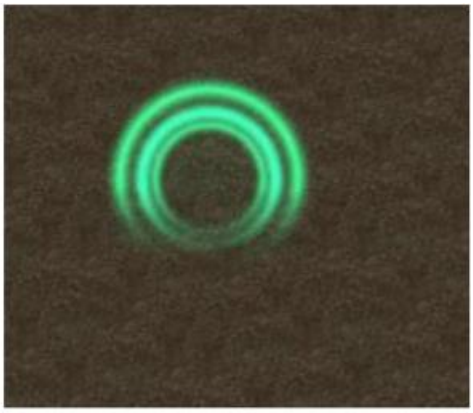


BARD

Restorative Serenade

Lament of Fate





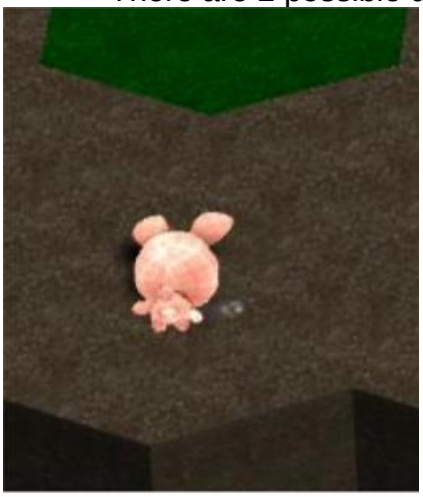
Viscous Mockery

-This could be done using an array of strings to be randomly called for more insults/taunts

-The display for this should be centered on middle of camera on the map not on the unit

Unit Death

-There are 2 possible death animations



Once all player’s units are defeated, victory or death is claimed and the game ends. The time to play is supposed to be around 15 minutes. The game logic is not 100% correct or well tested and will have to be reformatted, but the backbone of the game is there.

Technologies Used

Unity Game Editor 5.3.2 Personal
Java 1.8
C#
Git and Github
Android Studio
Netbeans 8
Visual Studio

Front-End: Client

Overview of the game client

The primary goal was to build a two-player, turn-based game in Unity that would implement a chess like strategy concept to the game to make players think critically and read the other players move. Keeping this in mind, the client team had to deal with a number of components such as the player and its opponent, unit placement, map creation, overall gameplay (position of units, manipulating health), player synchronization, communicating with the server and taking action based on its responses.

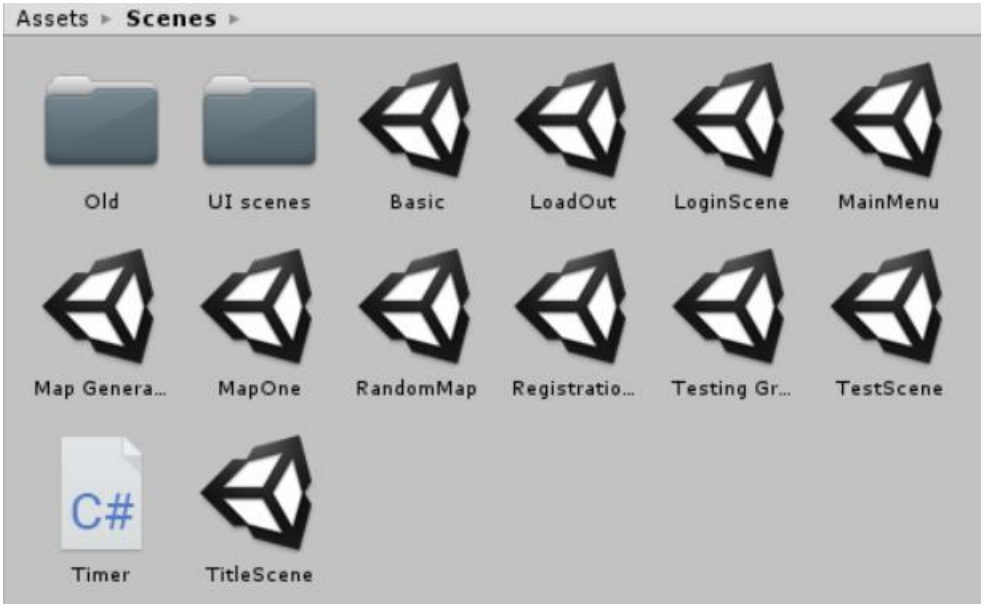
Unity client directory structure:

The assets associated with First Fantasy Wars have been isolated into the Assets directory.
Needs to be done: organize the scripts, materials, etc based on the scene name they are associated with to make it easier to keep track of:

First Fantasy Wars Unity client top level directory structure

Unity scenes, important GameObjects, components and scripts:

The game client in its current state consists of the following 11 scenes, 6 are primarily used:



Loadout, LoginScene, MainMenu, MapOne, RegistrationScene, and TitleScene are primarily used.

Title Scene:

- Canvas GameObject
    - Controller GameObject
        - MainMenu.c
            - GoToLoginScreen() - load login scene
            - GoToRegisterScreen() - load register scene

Login Scene:

- Canvas Gameobject
    - LoginController Gameobject
        - Login.c
            - LoginButton() - Verify username and password through data base and load the next scene, which is main menu where you pick what game mode you want
            - GoBackButton() - Go back to the title scene

Register Scene:

- Canvas GameObject
    - RegisterController
        - GameObject
            - CreateAcc.c
                - ConfirmButton() - Verify if all inputs are correct through database and load the next scene, which is login scene.

        - GoBackButton() - Go back to the title scene

Main Menu Scene:

- Canvas GameObject
    - Controller GameObject
        - OptionMenu.c
            - LoadOutScreen() - Load loadout screen (Scene where you choose unit and abilities)
            - Play() - Go directly to play if you have already chosen units and abilities in previous games.
            - LogOut() - Go back to title scene

Important! Look at StaticInfo.c - contains static data to use it to save data from LoadoutController.c and use it for gameplay scene.

StaticInfo.c:

- Important variables!
    - [SerializeField] Unit[] UnitChoices - Plug in a child class of Unit, must be in the order of melee/mage/bard. It is an array of unit choices indexed from 0 - 2.
    - LoadUnits() - Will try to read from file to see if it has any previous saved data, if not, then it will load saved data from the previous scene, which is Loadout scene. After it loaded the data, it will then initiate Unit objects from [SerializeField] Unit[] UnitChoices. It will load them based on the saved number from 0 - 2. 0 is melee, 1 is mage, and 2 is bard.

Loadout Scene:

- LoadOutScreen GameObject
    - GameController GameObject
        - LoadOutController.c
            - GoBackButton() - Load title scene
            - ChangeModel() - Takes an Unit class as an argument, each instance of an Unit object has a unique char number from 0 - 2. 0 is melee, 1 is mage, and 2 is bard. The numbers indicates the indexes of where the units will be located in the array when used to load units in the next scene. It will save that number in StaticInfo.c
            - CharSlot() - Takes an int as an argument, the int is to indicate which character slot the player has chosen. There are total 5 character slots, indexed from 0 - 4. So the int argument should be from 0 - 4. Also if the selected slot isn't null, then it will populate the information of that selected character slot on the menu boxes to display to user.
            - SkillSlotSelect() - Takes an int as an argument, the int is to indicate which available skill is selected. There are total of 5 available skills for each class. Each skill indexed from 0 - 4. Available skills are built inside each Unit object. Whichever

the skill is selected, it will save it to the StaticInfo.c as an chosen ability. It will simply save a number, which will be from 0 - 4.

- SelectChosenSkill() - Takes an int as an argument, the int is to indicate which slot of the chosen skill will be inserted to. The slot is indexed from 0 - 2.
- PopulateAvailableSkills() - Populate the game scene with available skills to choose from.
- PopulateChosenSkills() - If any skills are chosen from the previous game, it will populate the game scene with chosen skills, otherwise, it will show empty.
- EnableAvailableSkills() - If any skills are chosen, it will not be available to be chosen again. Therefore, if new skill is selected into the chosen slot, then call this method will repopulate the available skills to be available to be chosen again.
- ClearAvailableSkills() - Set all available skills to be available to choose
- ClearChosenSkills() - Clear the populating chosen skills with no text
- Done() - Load the gameplay scene with the saved data.
- WriteToFile() - Write all selected units and selected skills into a file. Each unit and ability are indicated as a number. Units from 0 - 2, abilities from 0 - 4. Each number indicate the slot number that of the character that are saved in the StaticInfo.c.
- ReadFromFile() - If data are saved from previous game, then calling this function will load the data from the previous saved. Otherwise, it will throw an exception and simply move on.
- Setup() - It will set loaded data to StaticInfo.c and store it for next scene use

Unit Mechanics:

Units have a base stat template with health, movement range, attack, defense, and three abilities.

Each unit also has room for an animator, and X,Y map coordinates.

The movement uses a list of tiles for pathfinding. As long as the list is not empty and the unit's destination is currently equal to its world position an element is popped from the list, a move method is called to move the unit from point A to point B and the unit's update method moves it towards the destination.

```
void Update ()
{
    //If Destination is reached
    if (destination.Equals (transform.position))
    {
        animator.SetBool ("Run", false); //Idle Animation
        if (movementPath.Count != 0)  //If there are more tiles in movement path we have to move to the next tile
        {
            this.Move (movementPath[0]); //Start moving to next tile
            movementPath.RemoveAt (0);   //Remove the next tile from the path
        }
    }

    else
    {
        //Move and rotate unit
        Vector3 differenceVector = (destination - transform.position).normalized;
        transform.rotation = Quaternion.Lerp (transform.rotation, Quaternion.LookRotation (differenceVector), 100f);
        animator.SetBool ("Run", true); //Run Animation
        Vector3 direction = destination - transform.position;
        Vector3 velocity = direction.normalized * speed * Time.deltaTime;

        //Do not let movement go over distance
        velocity = Vector3.ClampMagnitude (velocity, direction.magnitude);
        transform.Translate (velocity, Space.World);
    }

}
```

A list was used so that obstacles can be avoided and so that non-reachable tiles will not be treaded on.

There is a walking and idle animation that are used if the unit is moving or not moving.

When one unit is already present on a tile it becomes unwalkable so that no two units can occupy the same space.

The Action Controller:
This is responsible for finding tiles within a given range and A*pathfinding. These components will be used for movement, attacking, and retrieving the desired tiles.

There is a get successor function which is as follows:

```
//Returns possible successors with height constraint
public static List<HexagonTile> getSuccessors(HexagonTile currentTile, int maximumHeightDifference = 1)
{
    List <HexagonTile> successors = new List<HexagonTile> ();

    //The X, Y Coordinates for the current tile
    int x = currentTile.x;
    int y = currentTile.y;
    int offset = y % 2; //0 on even, 1 on odd

    //The X, Y coordinates to check for successors
    int[,] sucCoord = new int[6, 2]
    {
        { x - 1 + offset, y - 1 },
        { x + offset, y - 1 },
        { x - 1, y },
        { x + 1, y },
        {x - 1 + offset, y + 1},
        {x + offset, y + 1}
    };
```

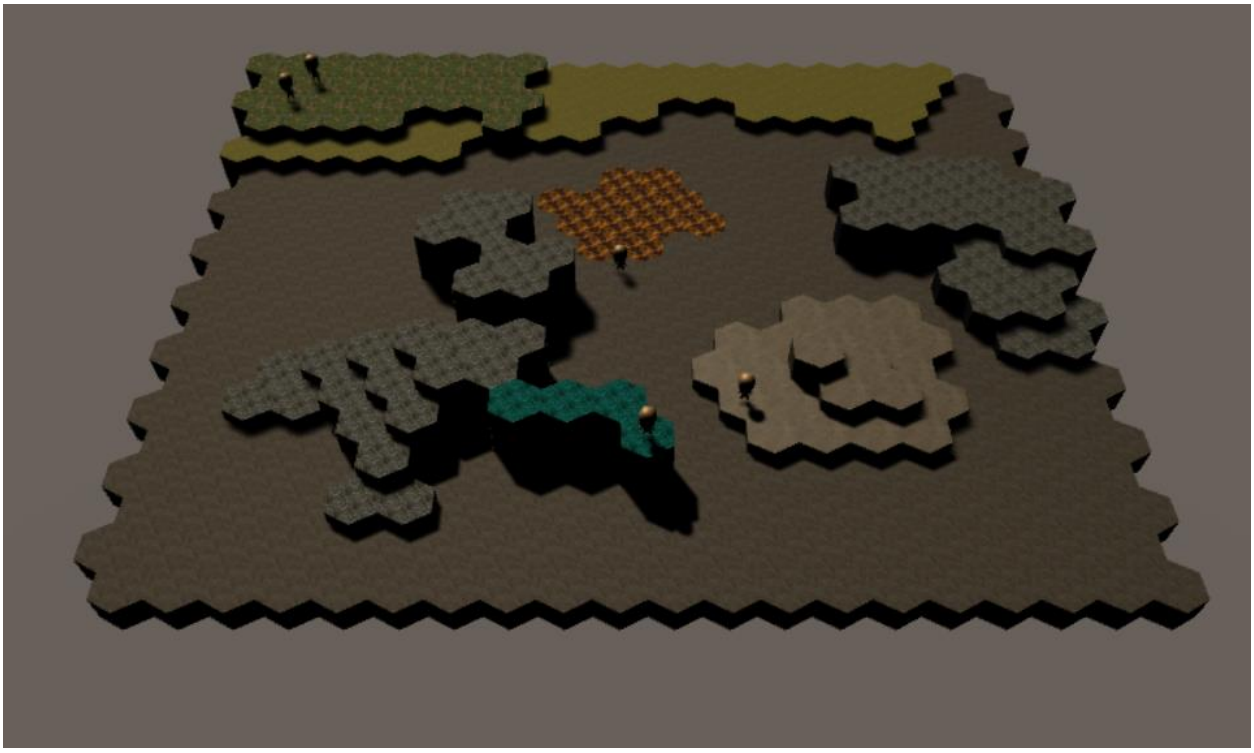
This is used within the search algorithms, A* for movement, and a search for tiles that are within a given range and height difference. These tiles are also checked to see if they are walkable.

The A* algorithm uses distance of the current tile and the destination tile along the horizontal plane as it's heuristic function.

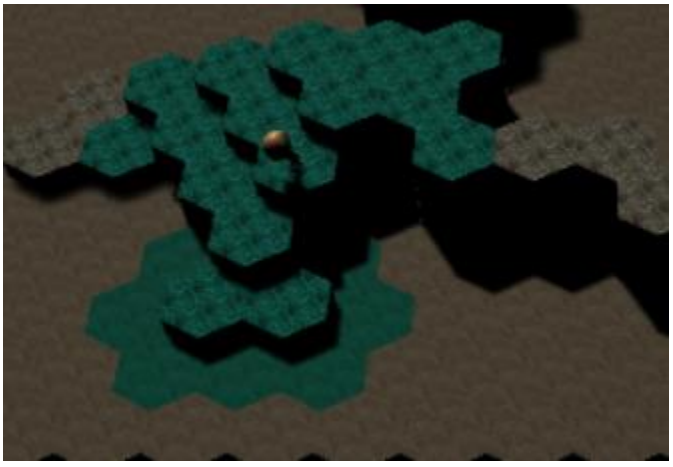
```
//A* movement for pathfinding
public static List<HexagonTile> unitPathfinding(HexagonTile startingTile, HexagonTile destinationTile, int maximumHeightDifference = 1)
{
    List<Node> visitted = new List<Node> ();
    Node currentNode = new Node (null, startingTile);

    SimplePriorityQueue<Node> fringe = new SimplePriorityQueue<Node> ();

    fringe.Enqueue(currentNode, currentNode.pathCost);
    while (fringe.Count > 0)
    {
        currentNode = fringe.Dequeue();
        if (!visitted.Contains (currentNode))
        {
            visitted.Add (currentNode);

            if (currentNode.hexagonTile == destinationTile)
                return NodeToHexagon(currentNode.nodePath());

            foreach (Node node in currentNode.expand())
            {
                if (!visitted.Contains (node))
                {
                    float distance = getDistance (startingTile, destinationTile);
                    if (node.hexagonTile.isWalkable && Mathf.Abs (node.hexagonTile.height - node.parent.hexagonTile.height) <= maximumHeightDifference)
                        fringe.Enqueue (node, node.pathCost + distance);
                }
            }
        }
    }
    return null;
}
```

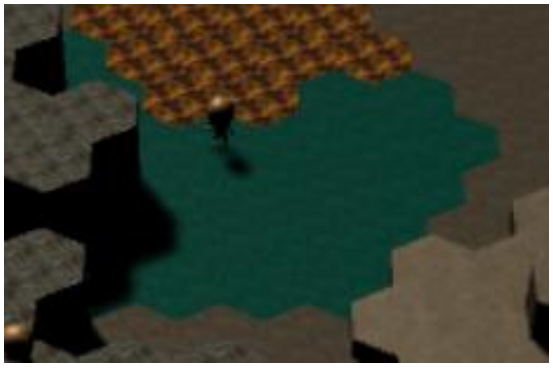
To return tiles within a certain range and maximum height difference the successor function is called until the desired node depth is reached (tracked within the node class), only grabbing successors that are within the maximum height difference constraint

Movement, notice how only tiles within the range of 3 and height difference of 1 are highlighted.



Movement paths with heights clearly demonstrated | Movement with unwalkable terrain (lava)





Showing it with newer model

The action controller also handles pathfinding for projectiles, which is similar to movement, except it can move on terrain that is not walkable (such as lava or other units).

Abilities:
Abilities are currently a MonoBehavior object. This should not be. Origionally they were scriptable objects but the loadout made that incompatible (even though it was explicitly stated to make loadout take scriptable objects not MonoBehavior).

When we initially made abilities they were made as ScriptableObjects so that they could be used with the new keyword and not need the creating of prefabs or the updating of prefabs on changes. This would also allow new unit types to be easily implemented in loadout if it was done correctly as the loadout should be able to choose from the available abilities of the given class.

Syntax for this execution should be similar as follows:

```
availableAbilities[0] = new ArcaneMissiles();
availableAbilities[1] = new Fireball();
availableAbilities[2] = new GlacialSpike();
availableAbilities[3] = new Poison();
```

--Within the mage class

Abilities work heavily on inheritance. There is a baseline ability class:

```
public abstract class Ability  : MonoBehaviour
{
    protected int abilityID;
    protected string abilityName;
    protected string description;
    protected int effectDuration = 0;
    protected int coolDown = 0;
    protected int range;
    protected int damageRadius;


    public Ability( string abilityName, int abilityID, int range, string description, int dmgRad )
    {
        this.abilityName = abilityName;
        this.abilityID = abilityID;
        this.range = range;
        this.description = description;
        damageRadius = dmgRad;
    }
```

*Damage radius should be replaced with effect radius

Projectile abilities inherit from this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ProjectileAbility : Ability
{
    public static Object projectilePrefab;// = Resources.Load("Visual/Projectiles/Fireball");
    public int damage;
    public string projectilePath;

    public ProjectileAbility(string projectilePath, int damage, int range, string abilityName, int abilityID, string description, int dmgrad)
        :base(abilityName, abilityID, range, description, dmgrad)
    {
        //projectilePrefab = Resources.Load(projectilePath);
        this.projectilePath = projectilePath;
        this.damage = damage;
        this.range = range;
    }

    public override bool activate(HexagonTile targetTile, Unit castingUnit)
    {
        projectilePrefab = Resources.Load(projectilePath);
        GameObject projectile = Instantiate(projectilePrefab) as GameObject;
        ImpactProjectile impactProjectile = projectile.GetComponent<ImpactProjectile>();
        impactProjectile.PlaceProjectile(castingUnit.currentTile);
        impactProjectile.Move(targetTile);
        return true;
    }

    public override void deactivate(Unit unitStats)
    {

    }
}
```

On target ability also inherits from this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class OnTargetAbility : Ability
{
    public string visualPath;
    public static Object visualPrefab;// = Resources.Load("Visual/Projectiles/Fireball");


    public OnTargetAbility(string visualPath, int range, string abilityName, int abilityID, string description) : base(abilityName, abilityID, range, description, 0)
    {
        this.visualPath = visualPath;
        this.range = range;
    }

    public override bool activate(HexagonTile targetTile, Unit castingUnit)
    {
        visualPrefab = Resources.Load(visualPath);
        GameObject projectile = Instantiate(visualPrefab) as GameObject;
        TargetedVisual targetVisual = projectile.GetComponent<TargetedVisual>();
        targetVisual.PlaceVisual(targetTile);
        return true;
    }

    public override void deactivate(Unit unitStats)
    {

    }
}
```

These classes should be refactored and have some attributes changed for future expansion, more detail in section 4) .

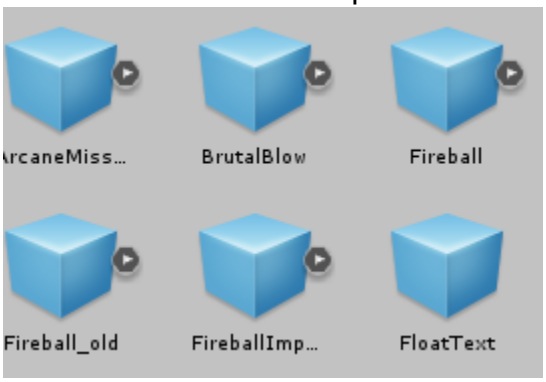
This allows for items that derive from it to be very short and concise.

Example: fireball is one line of code

```
public class Fireball : ProjectileAbility
{
    public Fireball() : base("Fireball", 40, 4, "Fireball", 5 , "Shoots a fireball!", 0) {}
}
```

Ability Visuals/ Ability Objects:

Abilities have visual components stored as prefabs:

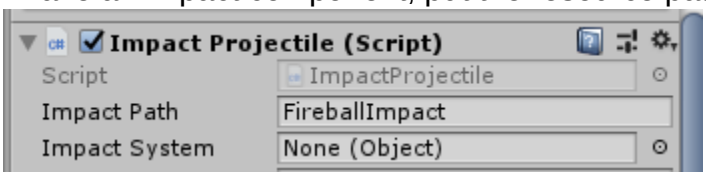


Fireball prefab:

Projectiles:
The visuals are found under resources/ visual / spells

Find a visual that looks nice.

Drag it to game world
-attach the ImpactProjectile script to it
-add sound by adding FMOD event emitters
-make an impact component, put the resource path for that in the impact path



Impact component:
Get the visual
-Attach impact script to it
--it will just work

Save the items to the correct path

Targeted Ability:
Grab visual, attach TargetedVisual script to it
Save it, done

Movement Code

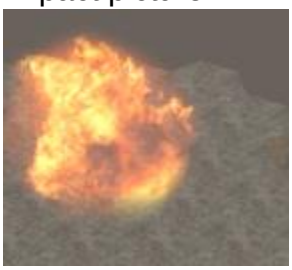
```
public bool Move(HexagonTile destinationTile) //Sets movement path
{
    this.destinationTile = destinationTile;
    this.movementPath = ActionController.projectilePathfinding(currentTile, destinationTile, 4);
    return true;
}
```

Placement and impact handling

```
public void PlaceProjectile( HexagonTile tile) //initial placement of projectile
{
    transform.position = tile.transform.position;
    transform.position = new Vector3(transform.position.x, transform.position.y + .5f * tile.height, transform.position.z);
    destination = transform.position;
    this.currentTile = tile;
}

public void Impact() //Called on impact
{
    Destroy(this.gameObject.GetComponent<ParticleSystem>());
    if(destinationTile.holdingUnit != null)
        destinationTile.holdingUnit.takeDamage(damage);
    impactSystem = Resources.Load(impactPath);
    GameObject projectile = Instantiate(impactSystem, gameObject.transform) as GameObject;
    hasImpacted = true;
}
```

They move to the destination when casted by a unit, are instantiated at the unit's position and on arrival it calls the impact which spawns the impact at the location. These self delete after the time (default 1.5 seconds) passed.

Impact picture:



On target visuals:

```
public void PlaceVisual(HexagonTile tile) //initial placement of projectile
{
    transform.position = tile.transform.position;
    transform.position = new Vector3(transform.position.x, transform.position.y + .5f * tile.height, transform.position.z);
    destination = transform.position;
    this.destinationTile = tile;
}

public void Update()
{
    persistTime -= Time.deltaTime;
    if (persistTime <= 0) Destroy(this.gameObject);
}
```

This just simply spawns the visual component at the location and self deletes when persist time finishes

Game Controllers

GameControlller is the baseline.
MouseController inherits from GameController.

Most items are static so they do not need an object reference to use or access and there will only ever be one controller for each element.

Units for players are held in an array. Turns alternate between players so that no one player can get many turns without being able to be reacted against. Player one moves the next alive/available unit in the array then player two moves their next alive/available unit throughout the course of the game.

The Map Controller has a map of the tiles stored as a two dimensional array.

Currently there is a mouse controller, which uses raycasting for tile selection and movement of units.

Turns are taken by parsing through the unit array, but currently turns end on movement since attacking has not been implemented yet.

The movement is handled by the action controller explained earlier.

Current unit is highlighted, depending on what the GUI component passes it phases change.

selectPhase: choose move, attack, wait
attackPhase: choose abilities
movePhase: movement choice

The gui passes the ability chosen to the attack phase for ability use and handles phase transitions. This is poor design, but it is what it is.

At each phase it should un-highlight everything, and handle some logic checks. Movement should only be able to occur once a turn and attacking ends turn.

A delay should likely be added after attacking before getting the next turn to allow the animations and damage to finish (if a unit will die from an attack, but damage has not been delivered yet it still gets a turn and this can be problematic).

Map Controller:

Parses the map and generates a two dimensional array used for ActionController components

```
public class MapController : MonoBehaviour {
    public static HexagonTile[,] hexagonTiles;
    public static HexagonTile currentTile;

    public int passedWidth;
    public int passedHeight;

    public static int mapWidth;
    public static int mapHeight;

    //Maps the Tiles
    void Start()
    {
        mapWidth = passedWidth;
        mapHeight = passedHeight;
        createTileMap ();
    }

    public static void createTileMap()
    {
        hexagonTiles = new HexagonTile[mapWidth, mapHeight];
        for (int width = 0; width < mapWidth; width++)
        {
            for (int height = 0; height < mapHeight; height++)
            {
                try{
                currentTile = GameObject.Find ("Hexagon" + width + "|" + height).GetComponent<HexagonTile> ();
                if (currentTile != null)
                    hexagonTiles [width, height] = currentTile;
                }catch(System.Exception exception){
                }
            }
        }
    }
```

Map Generator:

Generates a rectangular map used on parameters passed

```
void Start() //Creates everything
{
    AddGap();
    CalcStartPosition();
    CreateGrid();
}

void CalcStartPosition() //Finds starting position to center grid
{
    float offset = 0;
    if (gridHeight / 2 % 2 != 0) //On odd lines, offsets
        offset = hexWidth / 2;

    float x = -hexWidth * (gridWidth / 2) - offset;
    float z = hexHeight * .75f * (gridHeight / 2);

    startPosition = new Vector3 (x, 0, z);
}


Vector3 CalculateWorldPosition(Vector2 gridPosition) //Calculates placement
{
    float offset = 0;
    if (gridPosition.y % 2 != 0)
        offset = hexWidth / 2;

    float x = startPosition.x + gridPosition.x * hexWidth + offset;
    float z = startPosition.z - gridPosition.y * hexHeight * .75f;

    return new Vector3 (x, 0, z);
}

void AddGap() //Adds gaps
{
    hexWidth += hexWidth * gap;
    hexHeight += hexHeight * gap;
}


void CreateGrid() //Creates the grid, assigns identifiers
{
    for (int y = 0; y < gridHeight; y++)
    {
        for (int x = 0; x < gridWidth; x++)
        {
            Transform hexagon = Instantiate (hexagonTile) as Transform;
            Vector2 gridPosition = new Vector2 (x, y);
            hexagon.position = CalculateWorldPosition (gridPosition);
            hexagon.parent = this.transform;
            hexagon.name = "Hexagon" + x + "|" + y;

            tileProperties = hexagon.GetComponent<HexagonTile>();
            tileProperties.x = x;
            tileProperties.y = y;
        }
```

This lays hexagon tiles in rows.

Offsets every other row, then lays the next row. Repeats until it reaches the desired length.

Has ability to add gaps if desired.

The size is hard coded and extracted from blender.

Units:

```
public int attack = 20;
public int defense = 0;
public int move = 3;
public int xCoordinate;
public int yCoordinate;

[SerializeField] protected Ability[] abilities = new Ability[3];
[SerializeField] protected Ability[] availableAbilities = new Ability[5];
[SerializeField] private GameObject characterModel;
[SerializeField] private GameObject dummy;
//Booleans
public bool isPlaced = false;
public bool isMoving = false;
public bool isAttacking = false;
public bool isAlive = true;
```

```
public Unit(string charName, int maximumHealth, int currentHealth, int attack, int defense, int speed, int charNum)
{
    characterName = charName;
    maxHealth = maximumHealth;
    health = currentHealth;
    this.attack = attack;
    this.defense = defense;
    move = speed;
    charNumber = charNum;
    //Debug.Log("Calling " + characterName + " constructor and model number is " + charNumber);
}
```

Derived classes: Melee example

```
public class Melee : Unit
{
    public Melee():base("Melee", 120, 120, 20, 5, 3, 0)
    {
    }
```

On update ability moves to destination if available.

```
public bool Move(HexagonTile destinationTile)
{
    if(this.movable.Contains(destinationTile))
    {
        isMoving = true;
        currentTile.isWalkable = true;
        currentTile.holdingUnit = null;
        this.movementPath = ActionController.unitPathfinding(currentTile, destinationTile, 1);
        destinationTile.isWalkable = false;
        destinationTile.holdingUnit = this;
        xCoordinate = destinationTile.x;
        yCoordinate = destinationTile.y;

        return true;
    }
     return false;
}
```

move is initialized with the move method,

```
//Method for setting destination for movement
public void MoveOneStep( HexagonTile destinationTile)
{
    Vector3 hexagonLocation = destinationTile.transform.position;

    destination.x = hexagonLocation.x;
    destination.y = hexagonLocation.y;
    destination.y +=  .5f*(destinationTile.height);
    destination.z = hexagonLocation.z;


    currentTile = destinationTile;
}
```

destination is changed with moveOneStep

unit is placed with placeUnit for initial placement.

Attacking is done through the attack method

```
public bool Attack(HexagonTile targetTile, int abilityNumber)
{
    if (this.attackable.Contains(targetTile))
    {
        if (this.currentTile == targetTile) return false;
        //THEN CALL THE ATTACK ON THE TARGET TILE
        animator.SetTrigger("IsAttacking");
        transform.LookAt( new Vector3(targetTile.transform.position.x, transform.position.y + .5f, targetTile.transform.position.z ));
        abilities[abilityNumber].activate(targetTile, this);
        animator.SetTrigger("IsIdle");
        isAttacking = true;
        return true;
    }
    return false;
}
```

Movement and attacking have highlighting and retrieval of what is movable to or attackable through methods such as this

```
//*****************************************************
//Movement Tile Functions
//*****************************************************
public List<HexagonTile> GetMoveable()
{
    movable = ActionController.findMovable (currentTile, move);
    return movable;
}

public void HighlightMoveable()
{
    foreach (HexagonTile tile in movable)
        tile.Highlight (Color.cyan);
}

public void RemoveHighlightMovable()
{
    if (movable != null)
    {
        foreach (HexagonTile tile in movable)
            tile.removeHighlight();
    }
}
```

Damage taking and healing is done through these:

```
public void takeDamage(int rawDamage)
{
    if (protector != null)
    {
        protector.takeDamage(rawDamage);
        return;
    }
    health -= (rawDamage - defense);
    PopText("" + rawDamage);
    if (health <= 0)
    {
        float randomNumber = Random.Range(0, 1);
        {
            if (randomNumber < .5) animator.SetTrigger("Die1");
            else animator.SetTrigger("Die2");
        }
        isAlive = false;
    }
    animator.SetTrigger("IsDamaged");
    animator.SetTrigger("IsIdle");
}

public void heal(int healingDamage)
{
    if ((healingDamage + health) > maxHealth)
    {
        health = maxHealth;
    }
    else
    {
        health = healingDamage + health;
    }
    PopText("+" + healingDamage);
}
```

The update method is where tiles are moved to if in destination, and ability choice is updated for getting ability currently selected (used for seeing what tiles are attackable). If the game is over the winning team will dance, but this has not been tested well yet.

```
void Update ()
{
    if (health <= 0) isAlive = false;
    if(GameController.gameOver == true)
    {
        animator.SetTrigger("Win");
    }
    if (isPlaced == false && GameController.getGameStart() == true)
        placeUnit();

    if (isMoving == true)
    {
        //If Destination is reached
        if (destination.Equals(transform.position))
        {
            animator.SetBool("Run", false); //Idle Animation
            if (movementPath.Count != 0)
            {
                this.MoveOneStep(movementPath[0]);
                //Debug.Log("Removed" + movementPath[0].x + "|" + movementPath[0].y);
                movementPath.RemoveAt(0);
            }
            else
            {
                if (GameController.getAttackPhase())
                {
                    //abilities[GameController.abilityChosen].deactivate(this);
                    GameController.setAttackPhase(false);
                    GameController.setSelectPhase(true);
                    GameController.abilityChosen = -1;
                    isAttacking = false;
                }
                isMoving = false;
            }
        }
        else
        {
            Vector3 differenceVector = (destination - transform.position).normalized;
            transform.rotation = Quaternion.Lerp(transform.rotation, Quaternion.LookRotation(differenceVector), 100f);
            animator.SetBool("Run", true); //Run Animation
            Vector3 direction = destination - transform.position;
            Vector3 velocity = direction.normalized * speed * Time.deltaTime;

            //Do not let movement go over distance
            velocity = Vector3.ClampMagnitude(velocity, direction.magnitude);
            transform.Translate(velocity, Space.World);
        }
    }
}
```

UnitGUI:
Renders buttons that switch game phases

It is very bare bones right now and consists of only button renderings and scene transitions and should also be broken into clean little methods and linked with better looking GUI.

```
if(GameController.getGameStart() == false)
{
    if (GUI.Button(new Rect(10, 10, 60, 30), "START"))
    {
        GameController.StartGame();
        GameController.setSelectPhase(true);
        return;
    }

}


if (GameController.getGameStart())
{
    if (GameController.getSelectPhase())
    {
        if (GameController.hasMovedThisTurn == false)
        {
            if (GUI.Button(new Rect(10, 10, classButtonSizeX, classButtonSizeY), "M"))
                GameController.startMovePhase();
        }

        if (GUI.Button(new Rect(10, 45, classButtonSizeX, classButtonSizeY), "A"))
            GameController.startAttackPhase();

        if (GUI.Button(new Rect(10, 80, classButtonSizeX, classButtonSizeY), "X"))
            GameController.startSelectPhase();

        if (GUI.Button(new Rect(10, 115, classButtonSizeX, classButtonSizeY), "S"))
            GameController.getNextUnit();


    }
    else if (GameController.getMovePhase())
    {
```

Front-End: Networking

As of right now the client only connects to the server in the following scenes:

TitleScene:

Player connects to server using the servers ip address.

```
void Start () {
    if (atStartUp) {
        u1 = new User("35.197.31.116", 9252);
        atStartUp = false;
    }
}
```

LoginScene:

Player enters a valid username and password that is then referenced to the servers database to authenticate. If its valid the player is taken to the next scene.

```
public void Submit()
{
    user_name = user_name.Trim();
    password = password.Trim();

    if (user_name.Length == 0)
    {
        Debug.Log("User ID Required");
        GUI.FocusControl("username_field");
    }
    else if (password.Length == 0)
    {
        Debug.Log("Password Required");
        GUI.FocusControl("password_field");
    }
    else
    {
        if (MainMenu.u1.ss.login(user_name, password, 10000).succ)
        {
            SceneManager.LoadScene("Scenes/MainMenu");
        }
    }
}
```

RegistrationScene:

Player enters their email, username, and password they wish to associate to their new account. This is then added to the servers database to be used to login in the LoginScene.

```
    else
    {
        if (MainMenu.u1.ss.register(email, user_name, password, 10000).succ)
        {
            SceneManager.LoadScene("Scenes/LoginScene");
        }
    }
}
```

LoadoutScene:

Players customized loadout and unit abilities are send to the database. This part is still buggy where the server is not correctly taking the information and needs to be fixed before playing.

```
public void Done()
{
    if (!confirm())
    {
        Debug.Log("Something is not selected");
        return;
    }
    writeToFile();
    MainMenu.u1.ss.setUnits(unitId);
    MainMenu.u1.ss.setUnitsAbilities(unitId, abilityId);
    SceneManager.LoadScene("Scenes/MapOne");
}
```

Front-End: Models and Design

For this game we chose to go with a fantasy theme. Due to time constraints, the team decided to buy premade models that came with animations already built into them in order to save time as the project deadline came close. The set we decided on is called Yippy Kawaii, which included bears, cats, and bunnies with different texture options, as well as accessories and weapons. We also found a free 3D model of an electric guitar that we used for the bard class. Bears became the ‘melee’ class, cats became the ‘bard’ class, and bunnies became the ‘mage’ class. We kept the original color scheme of light and dark.

1. Melee Bears

For the melee class we went with the bear for a more aggressive character and added a hatchet for its weapon.

2. Mage Bunny

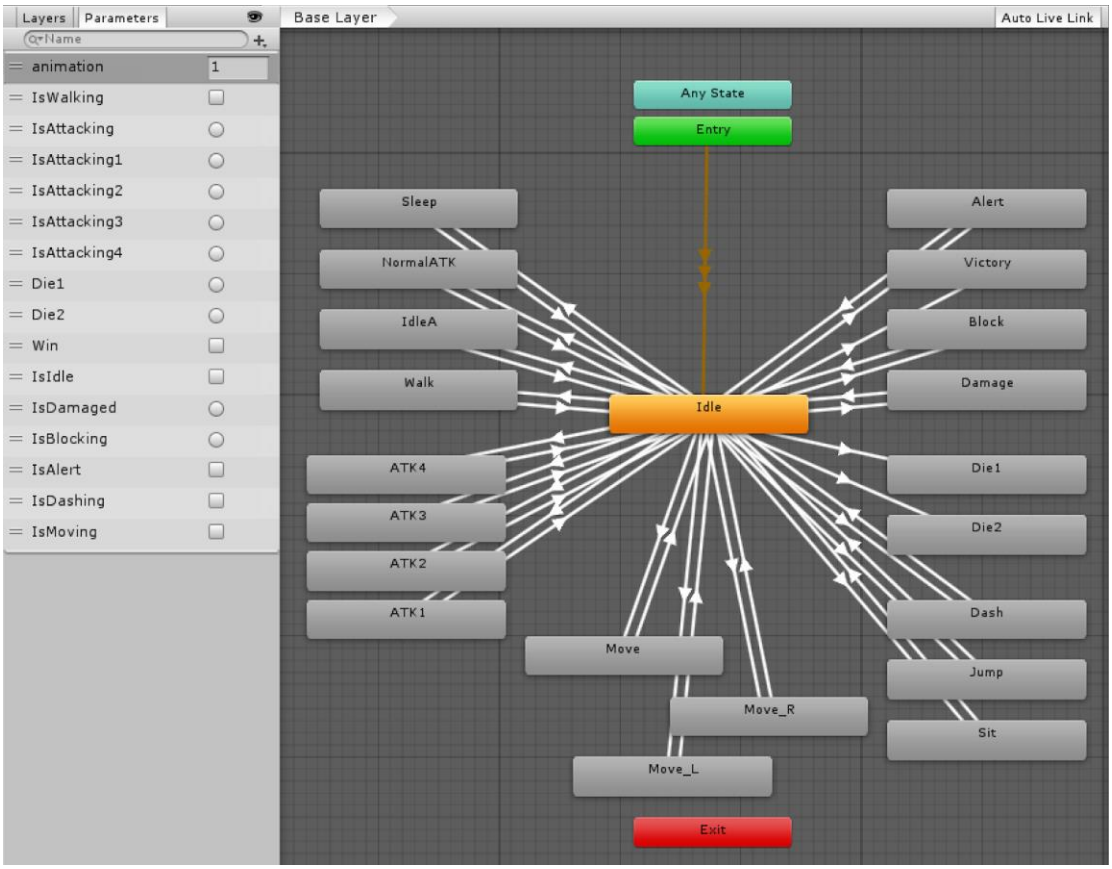
For the Mage class we went with the cute bunny for a more adorable but deadly character. The Mage uses a fish as its weapon.

3. Bard Cat

For the Bard class we went with the funny cat for a more fun but helpful character. The bard uses a guitar as its weapon.

Animations:

Animation controller



Although the animation controller was already created, it had an odd setup. There was an in between state from idle to something called ‘new state,’ which we decided to take out. This allowed idle to be connected to all the animations instead of having the ‘new state’ be what is connected to everything. Once we had that figured out, we created all the parameters needed for the animations we were choosing between to use. With these parameters created, we also tested the animations to see which ones would be useful.

Animation Testing

```
if (Input.GetKey("g"))
{
    if (!Dashing)
    {
        animator.SetBool("IsMoving", true);
        Dashing = true;
        animator.SetBool("IsIdle", false);
    }
    else
    {
        Dashing = false;
        animator.SetBool("IsMoving", false);
        animator.SetBool("IsIdle", true);
    }
}
```

This is a sample of the script we wrote to test the animations. When the key “g” is pressed, if the character is idle, it will cause the ‘dashing’ animation to play, but if the character is already ‘dashing’ then it will return to the idle state. The animations were set so that there were no interruptions, allowing them to play all the way through before returning to the idle state.

Audio Implementation

We used FMOD to integrate which made life easy for audio implementation, but we never found out how to properly call triggers inline and spent a few hours trying to find out why songs were not looping.

We want to thank our Audio Team for being so flexible and creative while creating the music and sfx for the game. Because we didn’t have a working demo until 3 weeks before the deadline, they had to create everything blind and they stuck the landing beautifully.

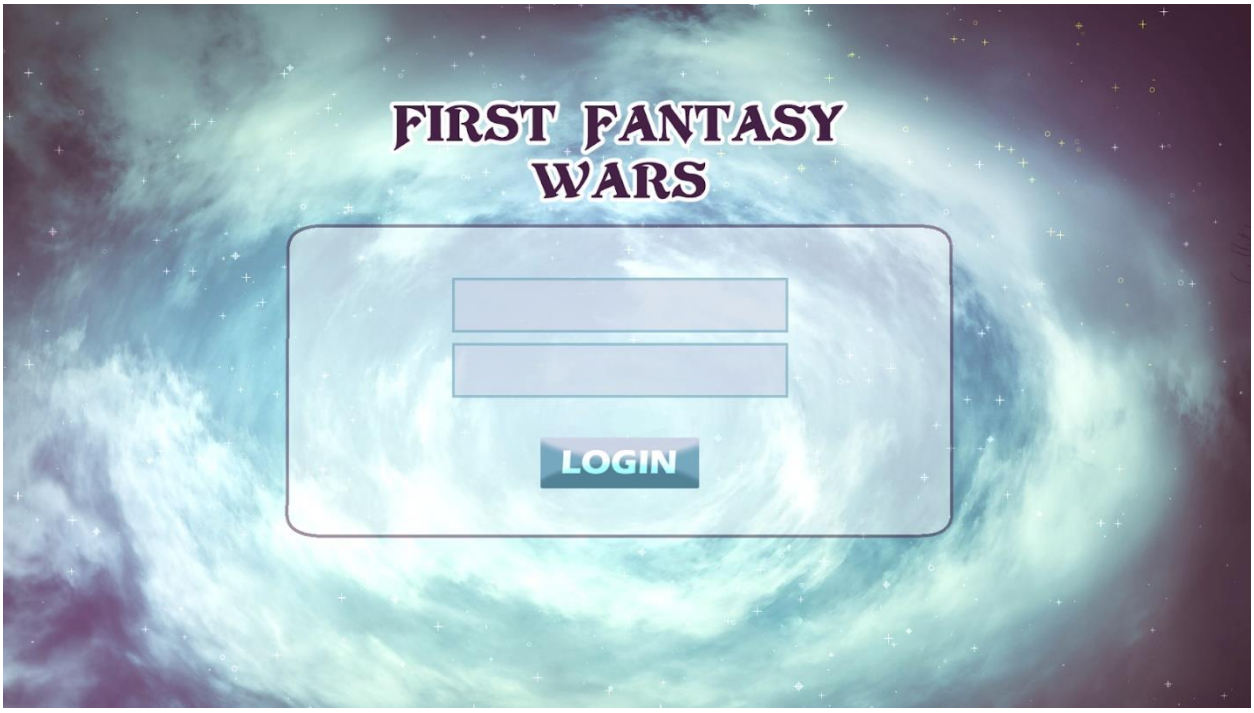
User Interface

Title Screen

This is the landing screen when the game is started. It will display the title and two buttons. One labeled “Register” and the other is “Login”. If a user does not have an account, they will not be able to log in. Therefore, they must first register an account. Both buttons will be clickable, regardless if the user has made an account.

When a user clicks on a button, the client will send a request to the server depending on the button that is clicked. If “Register” is clicked, the client will send a request to the server to route to the “Registration Screen”. If “Login” is clicked, then the client will request the server to route it to the “Login Screen”.

Login Screen



This is the “Login Screen” in which registered users shall be able to enter their username and password information. If the information is valid, then clicking the “Login” button will log in the player and transition them to the next scene.

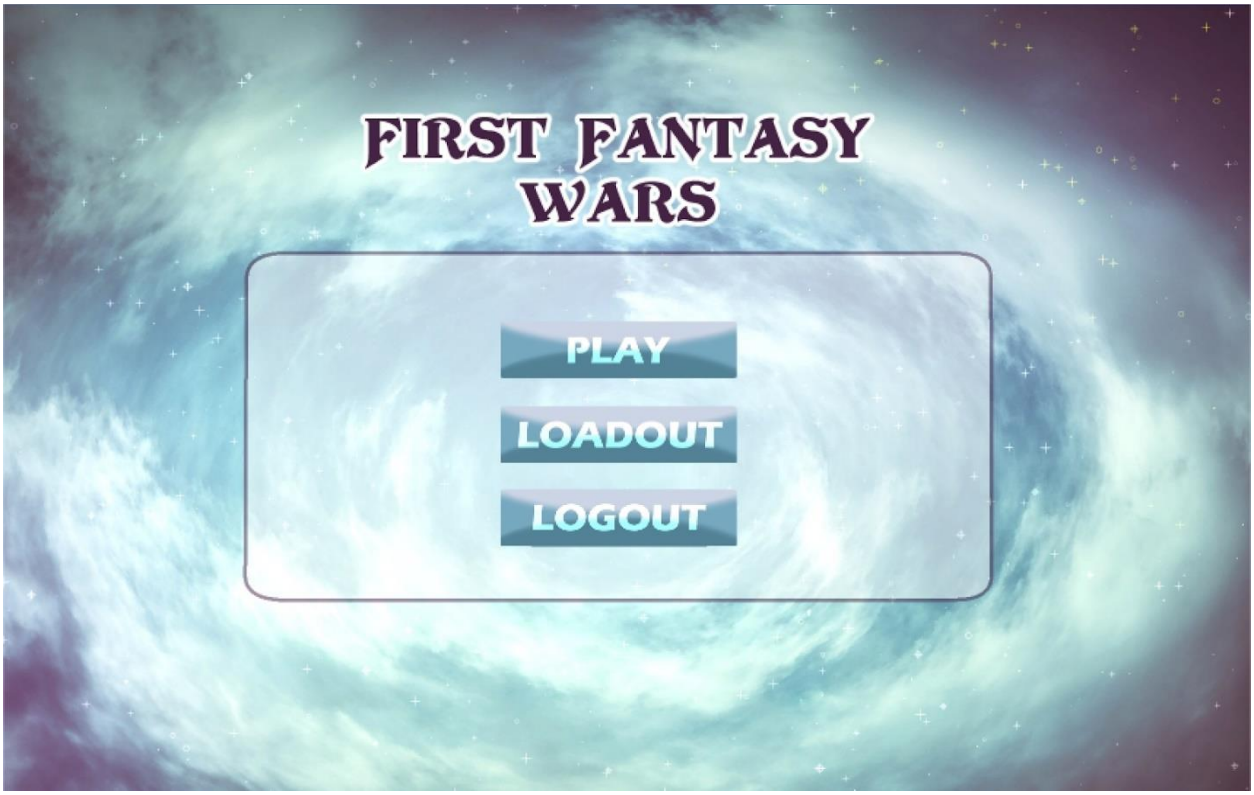
When a user has filled the fields and clicked the “Login” button, the client will first validate that all the fields have been filled. If the fields are all filled, the clicking the “Login” button will send a request along with the entered data to the server to check for valid fields. The server will check the database to see if the data that is entered matches the entered fields. On success, the server will send a validated response to the client to have it transition to the “Main Menu” screen.

Registration Screen

This is the registration screen for a user to register for a new account. In this screen the user will be able to fill the fields of “Email”, “Username”, & “Password”. There will also be a button, “Register”, which will confirm the entered fields.

When a user has filled the fields, and clicked the “Register” button, the client will first validate that all the fields have been filled with correct information. The client will then send a request along with the entered data to the server. This request will be asking the server to store the data into the database. If the unique fields of username and email do not exist in the database, then the server will send a response stating that the user is registered and will respond with a route back to the title menu.

Main Menu Screen



The “Main Menu Screen” will allow users to be able to have the option of choosing “Quickplay”, “Loadout”, or “Log Out”. Choosing the “Quickplay” option will match the player with another player to start a match. If they have not already created a team, once matched they will be able to transition to the “Loadout Screen”. They may also choose “Loadout” in order to pre-load a team, allowing them to play against others as soon as they are matched. Players will also have the option to “Log Out”, which will take them back to the “Title Menu”.

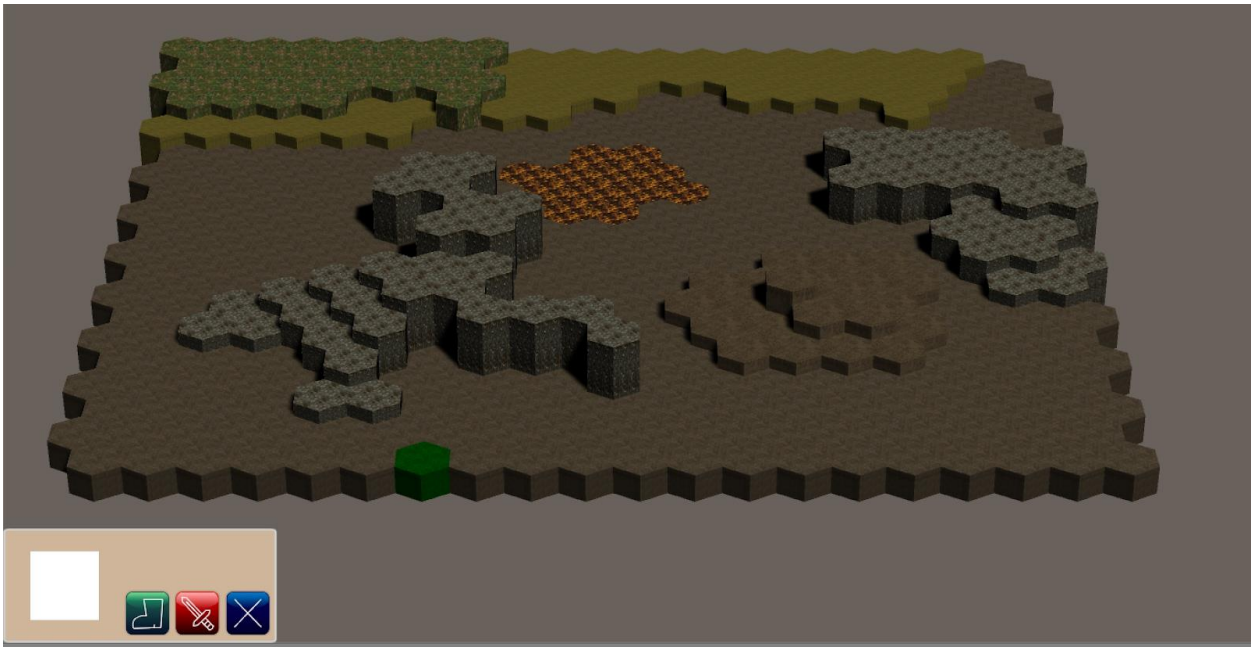
When a user has clicked on the “Quickplay” button, the client will send a request to the server to put the player in a “Match Queue”. The user will be placed at the back of the queue and will wait until their ID is popped from the queue along with one more player. After which, the server will send a response to the client to transition the player to the loadout screen.

Loadout Screen

This is the loadout screen, in which the user shall pick five units out of three possible classes which they will use for the game. Players will come to this screen through the main menu where they can choose to preset their team.

When a user has clicked on one of the class (“Melee” “Mage” or “Bard”), the client shall pull the skills for that unit type from the database and display it in the middle of the screen. If a user is satisfied with the skills of the class, they will then click on one of the spaces in their party to add a unit of that class. Once the user has chosen 5 units, they will click “Done”, which will send them back to the main menu screen.

GUI/Battle Screen



This is the battle screen in which the player faces off with another player in combat. The players will take turns moving their units and attacking each other. One player will win once the other player's units have all been eliminated.

The battle GUI was completed but not implemented due to shortage of time.

Mockups
Mockups

Character Design:

3D Models:

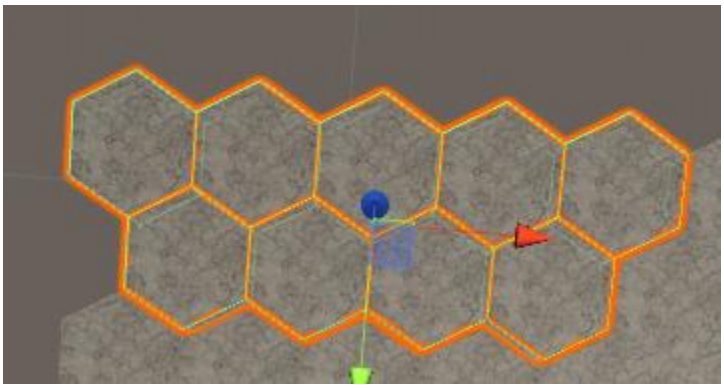
Mockups

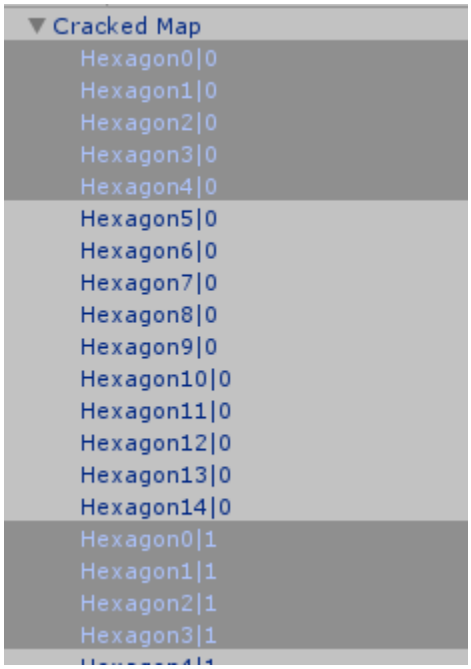
Character Design:

3D Models:

Client: Map Design

The game world is made out hexagons, which are organized into a two dimensional map of x,y coordinates. These hexagons were made from scratch in blender and textured using open source assets





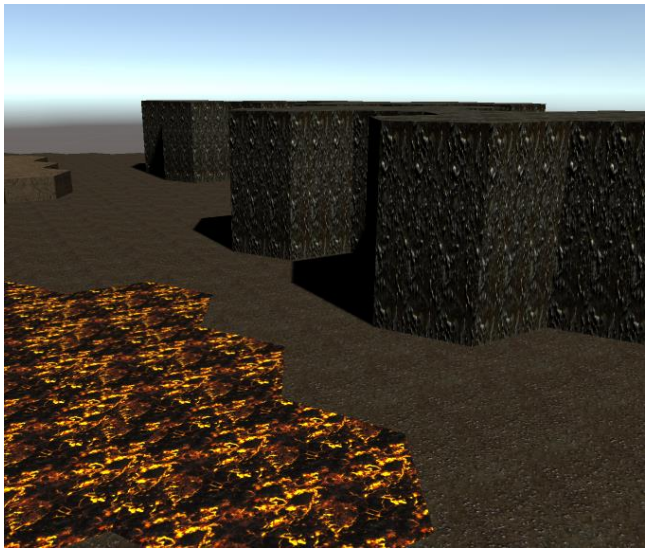
Left to right highlighted tiles:

0,0	1,0	2,0	3,0	4,0
0,1	1,1	2,1	3,1	

These coordinates are what is used for game locations since the game is based around the tiles.

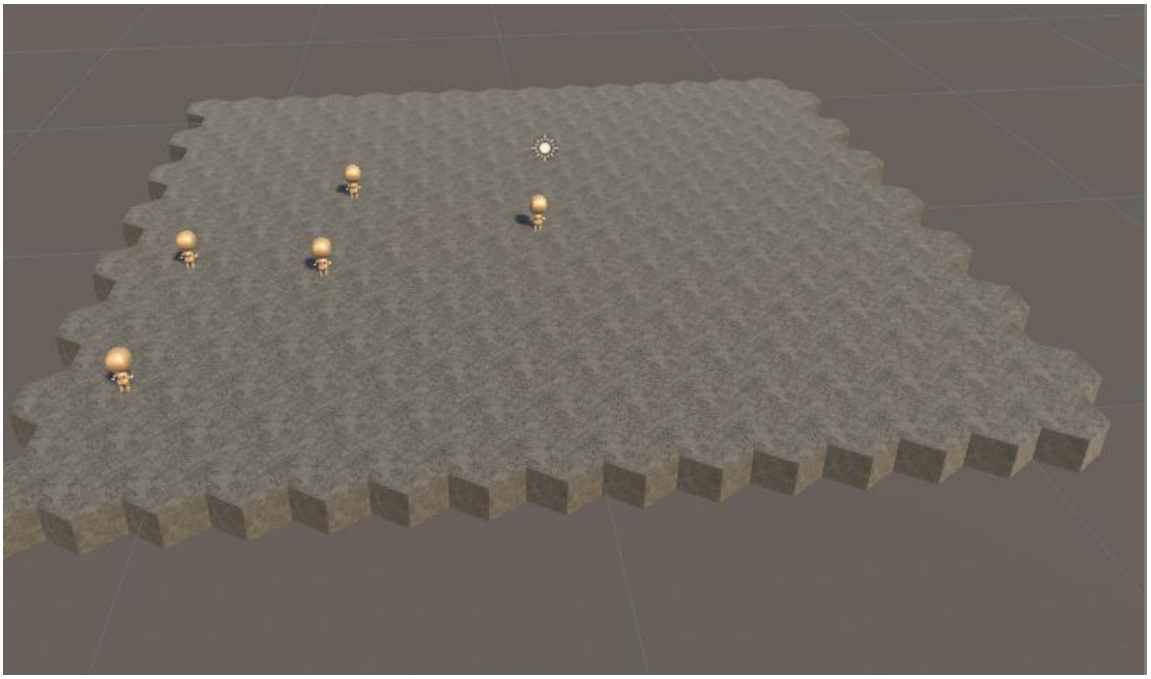
Units have X, Y coordinates as well: these correspond to the tile map coordinates.

Hexagon tiles can be walkable or not walkable and possess a height attribute as well. Tiles such as lava are not walkable and tiles with units present on top are also not walkable.

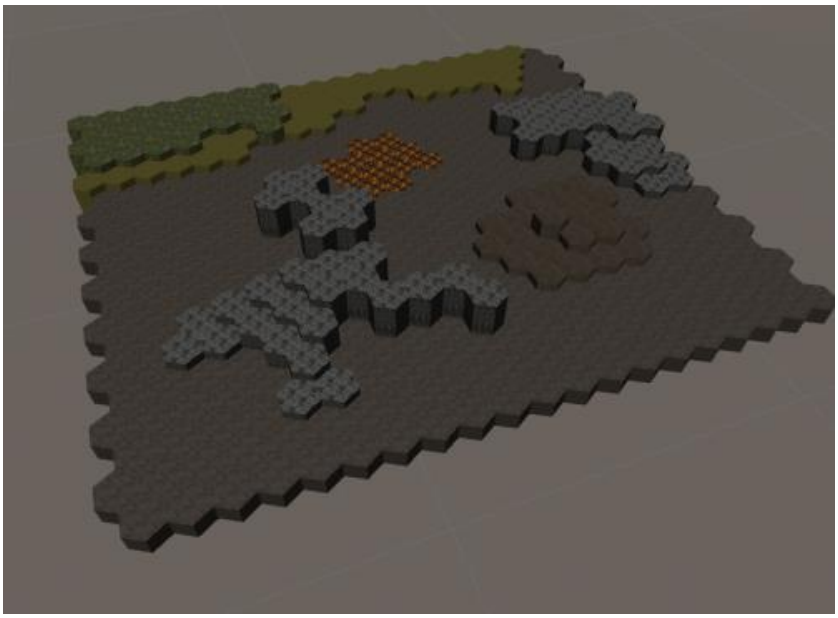


The movement takes into account height and whether tiles are able to walked on or not.

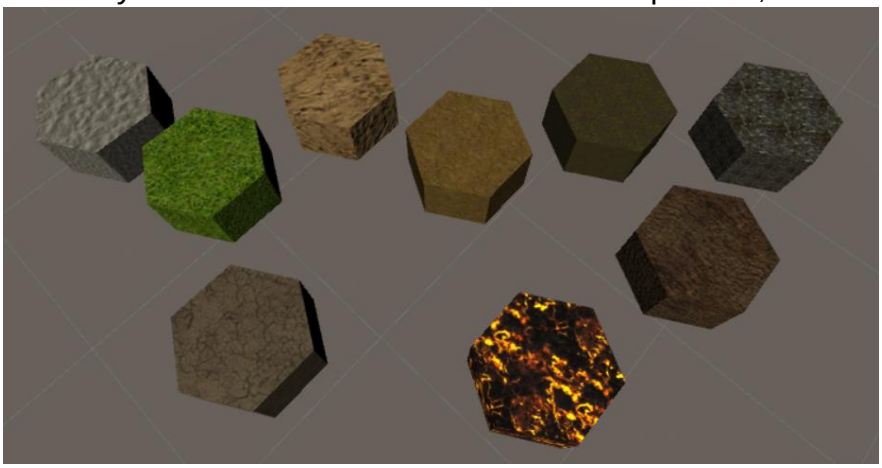
The basic foundation of the maps that I have worked on rely on a map generator script which lays tiles into a rectangle of set width and height. It places into rows and offsets the tiles by half of the tiles width on odd rows. This generates a simple map like the one shown below:



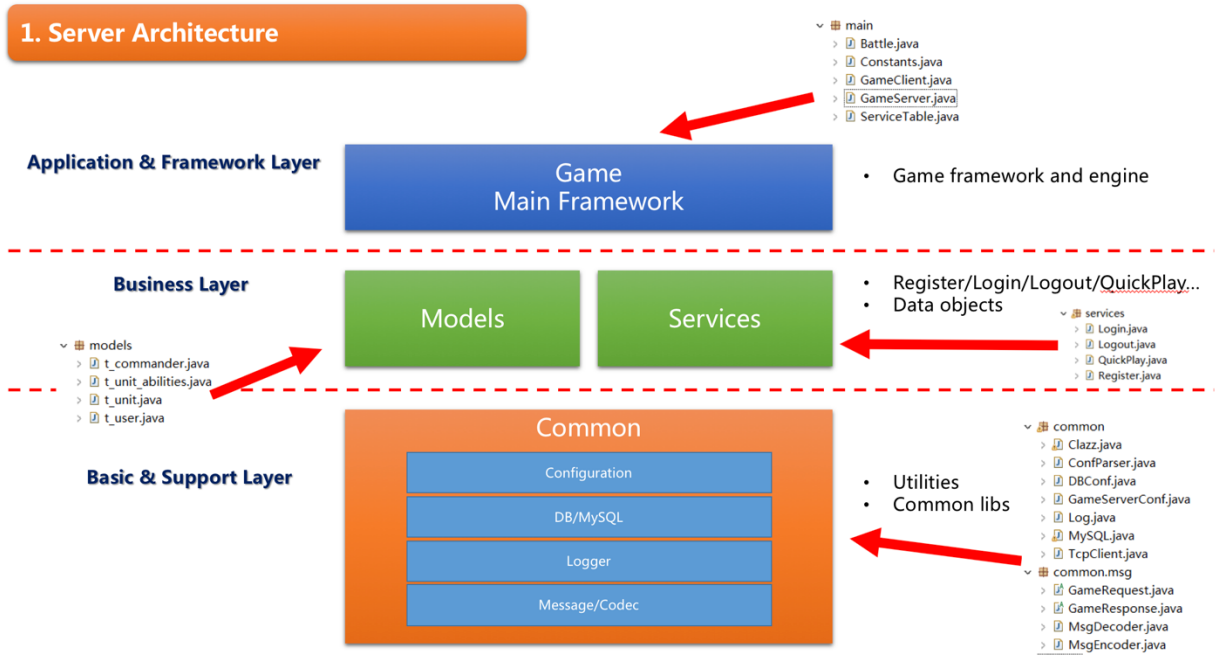
Extra details are added to the map and the height components were changed manually to get a map such as this:



Currently the tiles shown below are added as prefabs, more could be added later.



Server: Architecture and Testing



Register a new account, blocked and wait until there is any server response or time is up.

C# Syntax

```
result register(string email, string usr, string passwd, int timeout)
```

Parameters

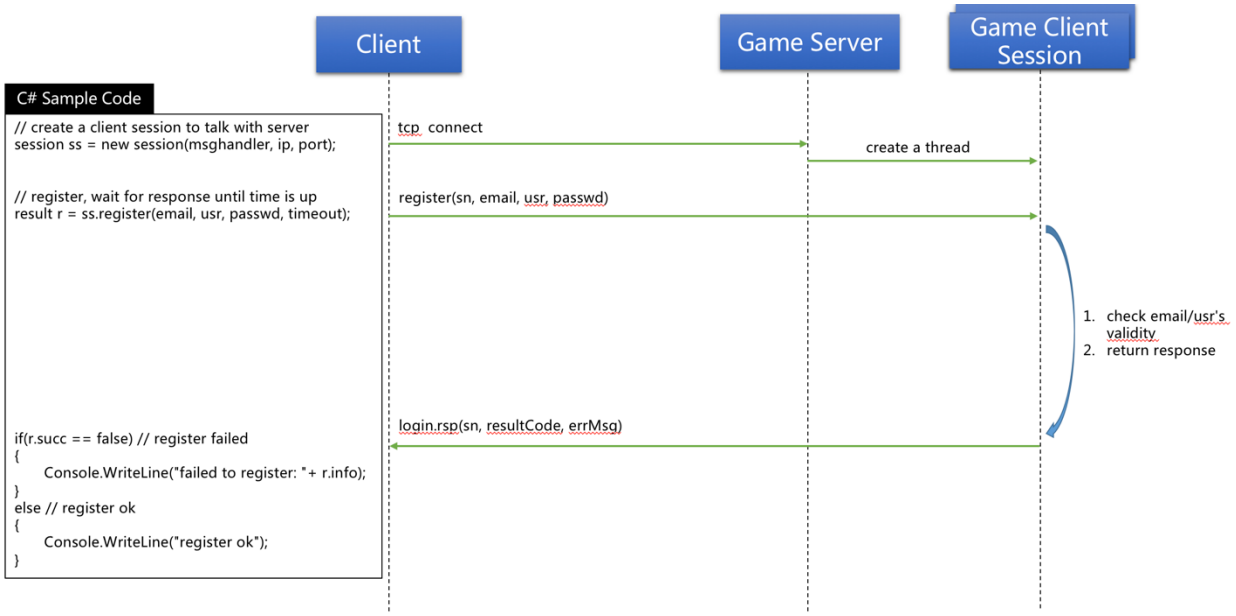
email: user's email, unique identity
usr: account's name that will be registered
passwd: new account's password
timeout: the timemout time if not any server responses

Return Values

```
public class result
{
    public bool succ;  // Specified the operation's result is successful or not: true for sucessful, false for not.
    public string info; // Filled with the information why the operation is failed. This value is set to empty string, if succ is true.
}
```

Remarks

If register is done successfuy, no need sending login again.



Account login to the server, wait until there is a server's response or time 's up.

C# Syntax

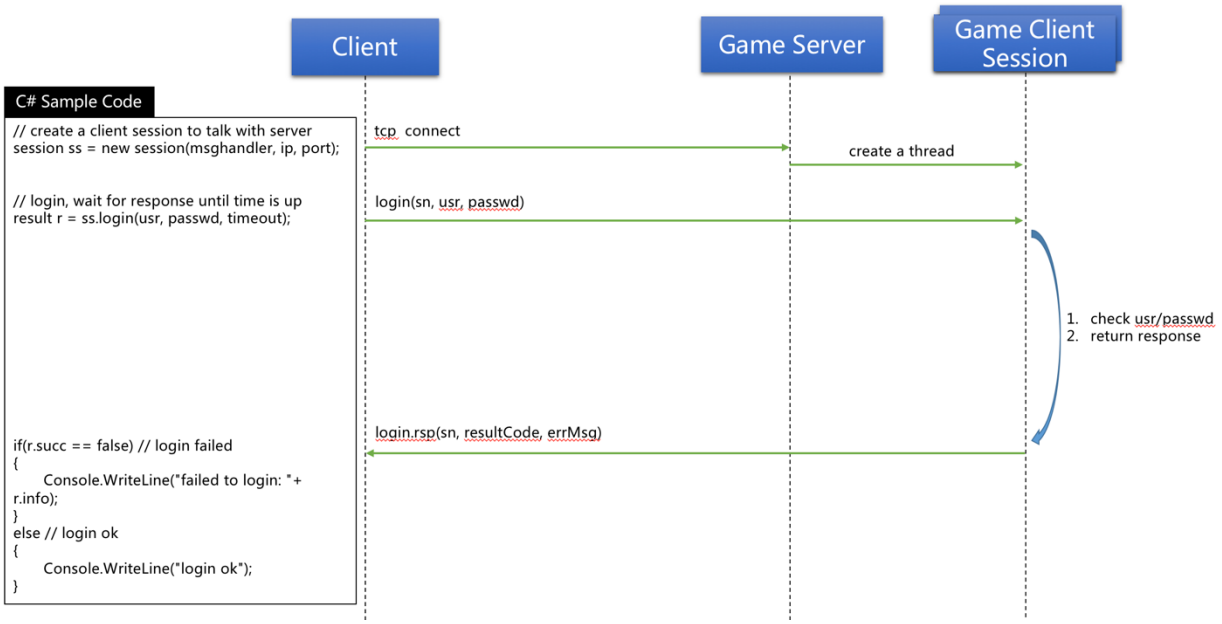
```
result login(string usr, string passwd, int timeout /* ms */);
```

Parameters

usr: the account's name
passwd: the account's password
timeout: the timemout time if not any server responses

Return Values

```
public class result
{
    public bool succ;  // Specified the operation's result is successful or not: true for sucessful, false for not.
    public string info; // Filled with the information why the operation is failed. This value is set to empty string, if succ is true.
}
```



Account logout from the server, blocked and wait until there is a server's response or time 's up.

C# Syntax

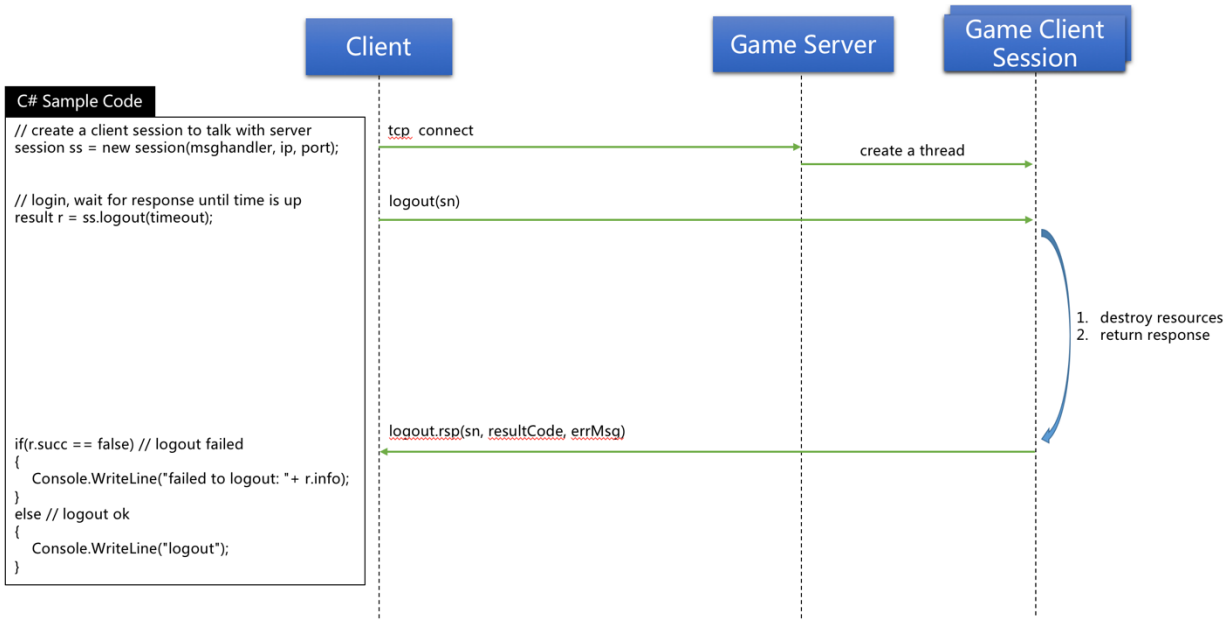
```
result login(int timeout /* ms */);
```

Parameters

timeout: the timemout time if not any server responses

Return Values

```
public class result
{
    public bool succ;  // Specified the operation's result is successful or not: true for sucessful, false for not.
    public string info; // Filled with the information why the operation is failed. This value is set to empty string, if succ is true.
}
```



User selects a commander and send a message to server, non-blocked and wait no responses at once. The server will reply the response back to the request , and then notify the opponent that which commander his or her opponent selects.

C# Syntax

```
result setCommander(long commanderId);
```

Parameters

commanderId: the id for commander

Return Values

```
public class result
{
    public bool succ;  // Specified the operation's result is successful or not: true for sucessful, false for not.
    public string info; // Filled with the information why the operation is failed. This value is set to empty string, if succ is true.
}
```

*This interface is non-blocked. And the possible error can be "connection lost" or "network error" .

A class to help decode SetUnits's response, setting the value of 'MY' verify result for the 5 unit ids.

C# Syntax

```
SetUnitsRsp t = new SetUnitsRsp(rsp);
```

Parameters

rsp: raw tcp response of SetUnits

Return Values

```
public class SetUnitsRsp
{
    public int[] check = {0, 0, 0, 0, 0}; // Specified the verify results.
    public bool succ;  // Specified the operation's result is successful or not: true for sucessful, false for not.
    public string info; // Filled with the information why the operation is failed. This value is set to empty string, if succ is true.
}
```

*The possible error can be "illegal operation, no matched battle" or "invalid units".

Synchronize the player's action to the other player, including the movement of unit, damage to the opponent's units

C# Syntax

```
result gameAction(long curUnit, int dstX, int dstY, long[] units, int[] damages)
```

Parameters

curUnit: current unit id
dstX: the target coordinate X, if not moved, set it to -1
dstY: the target coordinate Y, if not moved, set it to -1
units: the opponent's units' id list
damages: the damages to the opponent's **units,** if no damage, set all five values to zero.

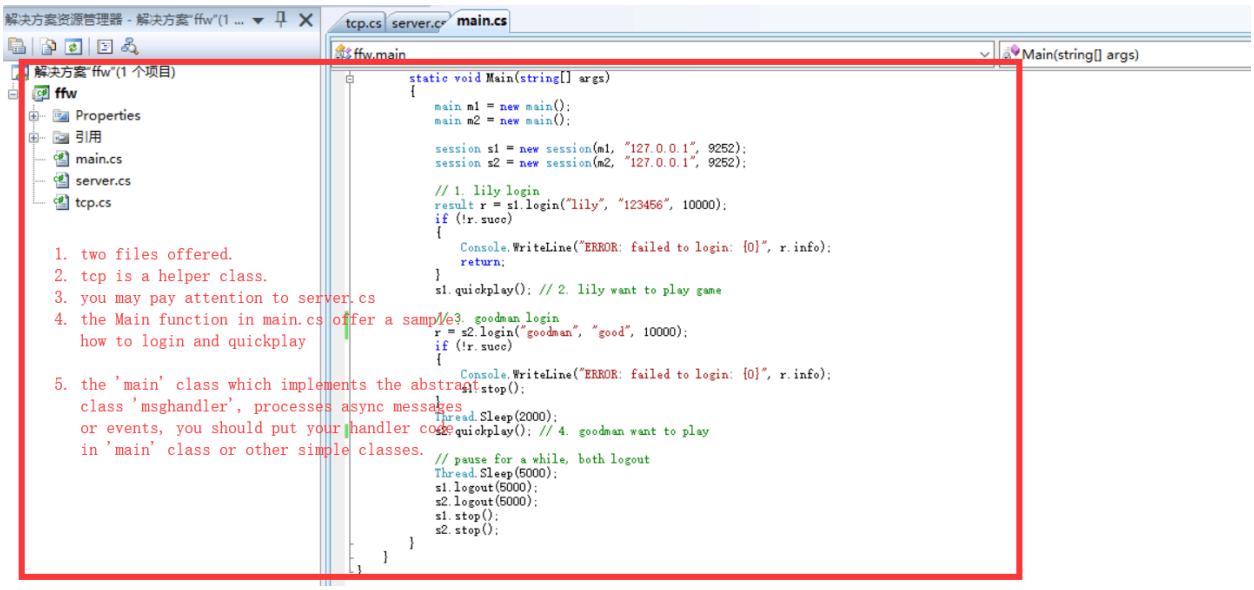
Return Values

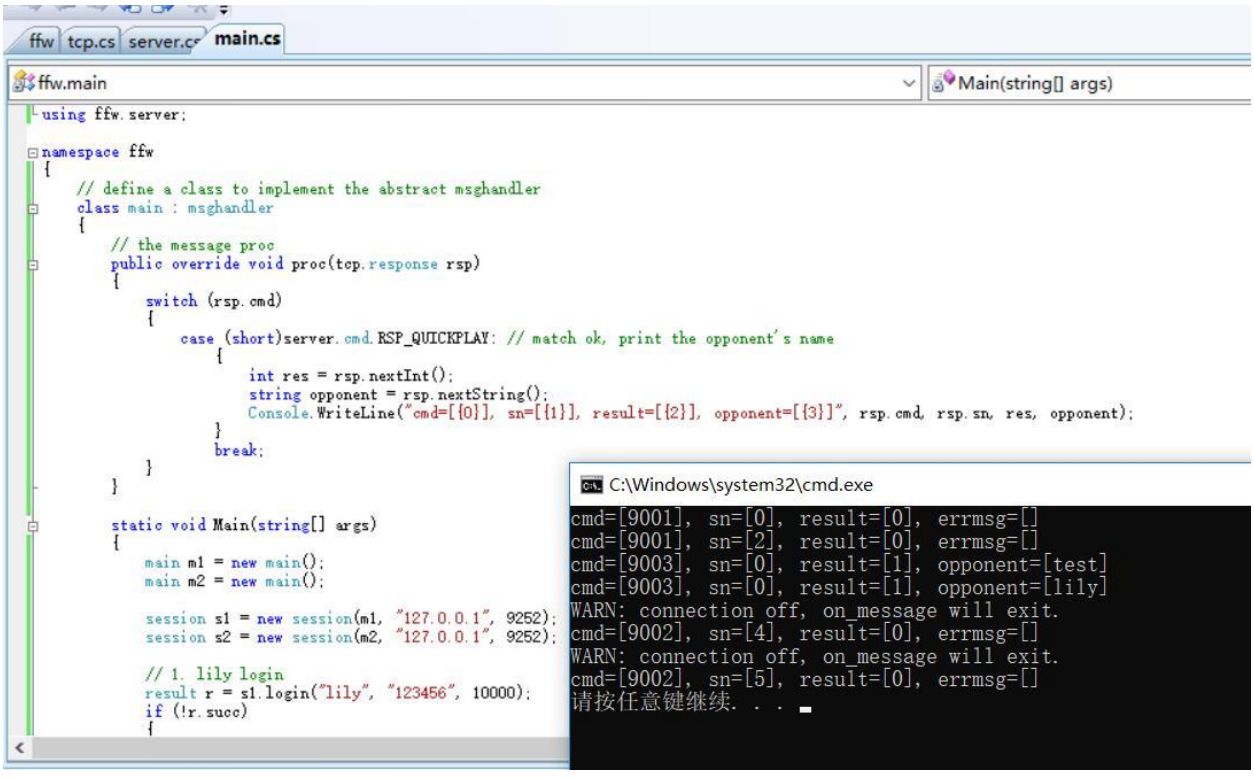
```
public class result
{
    public bool succ;  // Specified the operation's result is successful or not: true for sucessful, false for not.
    public string info; // Filled with the information why the operation is failed. This value is set to empty string, if succ is true.
}
```

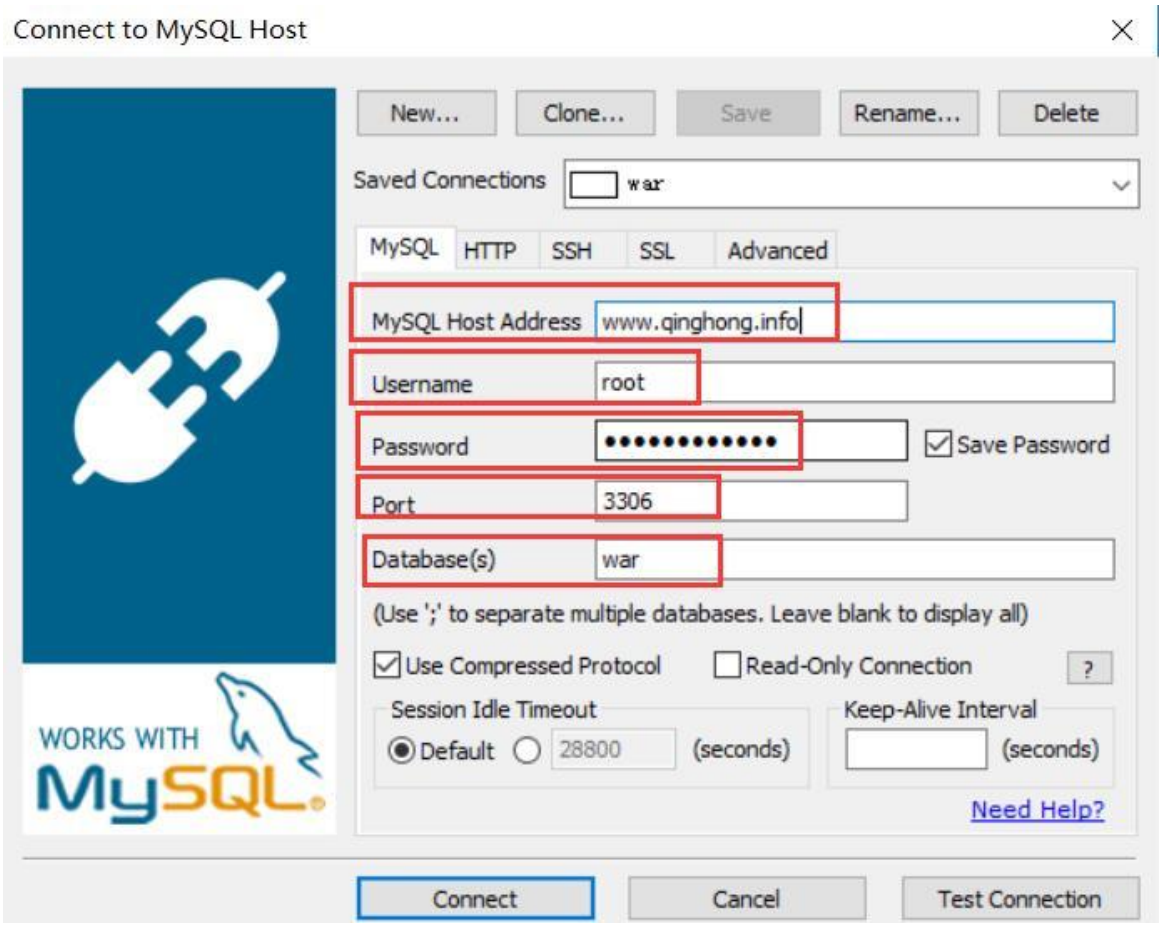
*This interface is non-blocked. And the possible error can be "connection lost" or "network error".

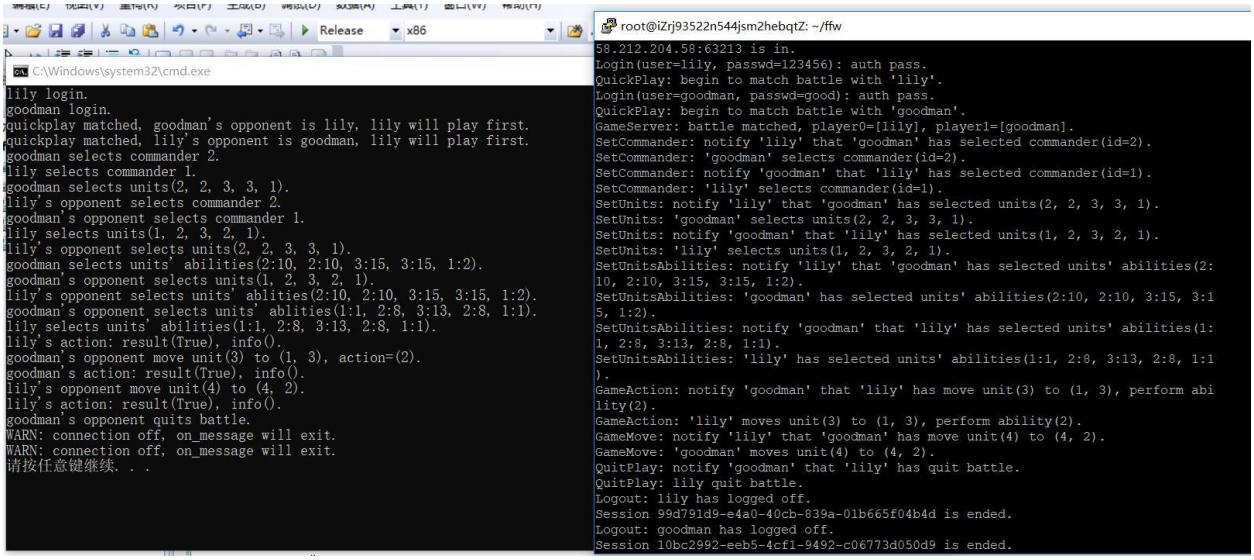
Server Testing:

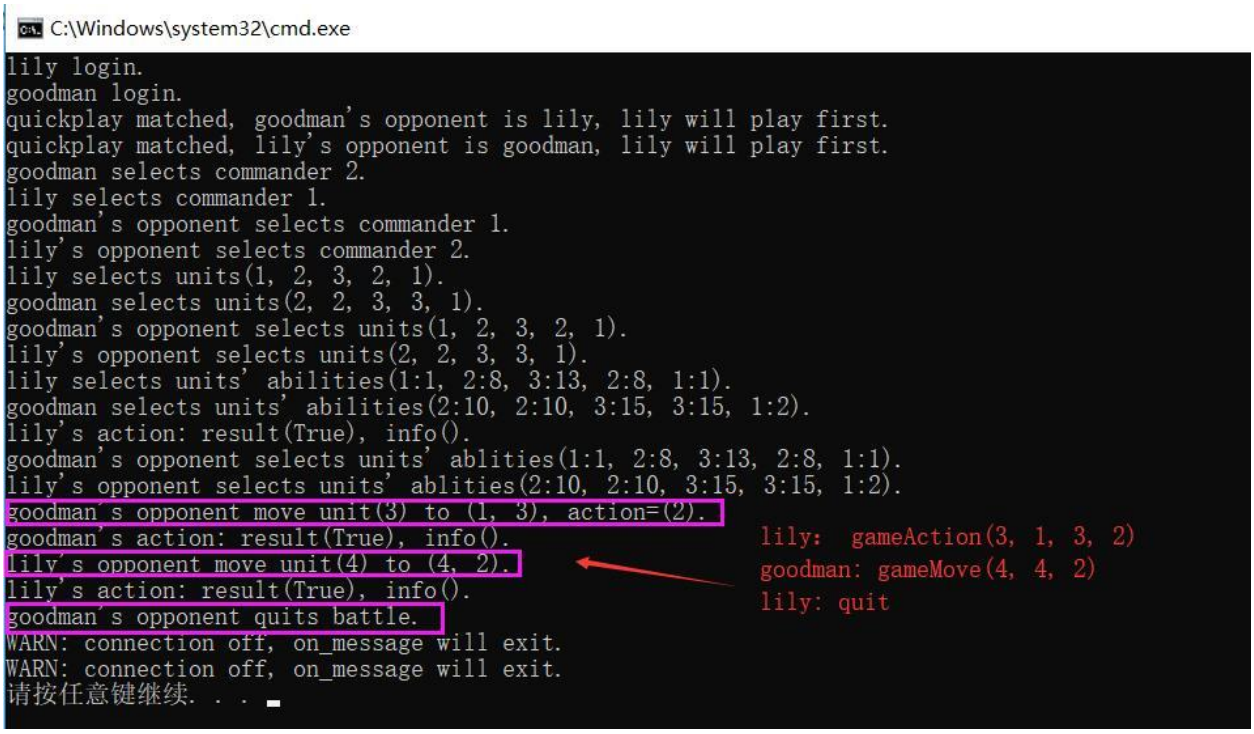
The team member in charge of writing the server code tested it by renting a web server to runt he code and used MySQL Workbench to test the database.


解决方案资源管理器 - 解决方案"ffw"(1 ...
解决方案"ffw"(1 个项目)
ffw
Properties
引用
main.cs
server.cs
tcp.cs
1. two files offered.
2. tcp is a helper class.
3. you may pay attention to server.cs
4. the Main function in main.cs offer a sample
how to login and quickplay
5. the 'main' class which implements the abstract
class 'msghandler', processes async messages
or events, you should put your handler code
in 'main' class or other simple classes.
tcp.cs
server.cs
main.cs
ffw.main
Main(string[] args)
static void Main(string[] args)
{
main m1 = new main();
main m2 = new main();
session s1 = new session(m1, "127.0.0.1", 9252);
session s2 = new session(m2, "127.0.0.1", 9252);
// 1. lily login
result r = s1.login("lily", "123456", 10000);
if (!r.succ)
{
Console.WriteLine("ERROR: failed to login: {0}", r.info);
return;
}
s1.quickplay(); // 2. lily want to play game
// 3. goodman login
r = s2.login("goodman", "good", 10000);
if (!r.succ)
{
Console.WriteLine("ERROR: failed to login: {0}", r.info);
s1.stop();
}
Thread.Sleep(2000);
s2.quickplay(); // 4. goodman want to play
// pause for a while, both logout
Thread.Sleep(5000);
s1.logout(5000);
s2.logout(5000);
s1.stop();
s2.stop();
}
}
}


ffw
tcp.cs
server.cs
main.cs
ffw.main
Main(string[] args)
using ffw.server;
namespace ffw
{
// define a class to implement the abstract msghandler
class main : msghandler
{
// the message proc
public override void proc(tcp.response rsp)
{
switch (rsp.cmd)
{
case (short)server.cmd.RSP_QUICKPLAY: // match ok, print the opponent's name
{
int res = rsp.nextInt();
string opponent = rsp.nextString();
Console.WriteLine("cmd=[{0}], sn=[{1}], result=[{2}], opponent=[{3}]", rsp.cmd, rsp.sn, res, opponent);
}
break;
}
}
static void Main(string[] args)
{
main m1 = new main();
main m2 = new main();
session s1 = new session(m1, "127.0.0.1", 9252);
session s2 = new session(m2, "127.0.0.1", 9252);
// 1. lily login
result r = s1.login("lily", "123456", 10000);
if (!r.succ)
C:\Windows\system32\cmd.exe
cmd=[9001], sn=[0], result=[0], errmsg=[]
cmd=[9001], sn=[2], result=[0], errmsg=[]
cmd=[9003], sn=[0], result=[1], opponent=[test]
cmd=[9003], sn=[0], result=[1], opponent=[lily]
WARN: connection off, on_message will exit.
cmd=[9002], sn=[4], result=[0], errmsg=[]
WARN: connection off, on_message will exit.
cmd=[9002], sn=[5], result=[0], errmsg=[]
请按任意键继续. . .


Connect to MySQL Host
New...
Clone...
Save
Rename...
Delete
Saved Connections
war
MySQL
HTTP
SSH
SSL
Advanced
MySQL Host Address
www.qinghong.info
Username
root
Password
Save Password
Port
3306
Database(s)
war
(Use ';' to separate multiple databases. Leave blank to display all)
Use Compressed Protocol
Read-Only Connection
?
Session Idle Timeout
Default
28800
(seconds)
Keep-Alive Interval
(seconds)
Need Help?
WORKS WITH
MySQL
Connect
Cancel
Test Connection


Release
x86
C:\Windows\system32\cmd.exe
lily login.
goodman login.
quickplay matched, goodman's opponent is lily, lily will play first.
quickplay matched, lily's opponent is goodman, lily will play first.
goodman selects commander 2.
lily selects commander 1.
goodman selects units(2, 2, 3, 3, 1).
lily's opponent selects commander 2.
goodman's opponent selects commander 1.
lily selects units(1, 2, 3, 2, 1).
lily's opponent selects units(2, 2, 3, 3, 1).
goodman selects units' abilities(2:10, 2:10, 3:15, 3:15, 1:2).
goodman's opponent selects units(1, 2, 3, 2, 1).
lily's opponent selects units' ablities(2:10, 2:10, 3:15, 3:15, 1:2).
goodman's opponent selects units' ablities(1:1, 2:8, 3:13, 2:8, 1:1).
lily selects units' abilities(1:1, 2:8, 3:13, 2:8, 1:1).
lily's action: result(True), info().
goodman's opponent move unit(3) to (1, 3), action=(2).
goodman's action: result(True), info().
lily's opponent move unit(4) to (4, 2).
lily's action: result(True), info().
goodman's opponent quits battle.
WARN: connection off, on_message will exit.
WARN: connection off, on_message will exit.
请按任意键继续. . .
root@iZrj93522n544jsm2hebqtZ: ~/ffw
58.212.204.58:63213 is in.
Login(user=lily, passwd=123456): auth pass.
QuickPlay: begin to match battle with 'lily'.
Login(user=goodman, passwd=good): auth pass.
QuickPlay: begin to match battle with 'goodman'.
GameServer: battle matched, player0=[lily], player1=[goodman].
SetCommander: notify 'lily' that 'goodman' has selected commander(id=2).
SetCommander: 'goodman' selects commander(id=2).
SetCommander: notify 'goodman' that 'lily' has selected commander(id=1).
SetCommander: 'lily' selects commander(id=1).
SetUnits: notify 'lily' that 'goodman' has selected units(2, 2, 3, 3, 1).
SetUnits: 'goodman' selects units(2, 2, 3, 3, 1).
SetUnits: notify 'goodman' that 'lily' has selected units(1, 2, 3, 2, 1).
SetUnits: 'lily' selects units(1, 2, 3, 2, 1).
SetUnitsAbilities: notify 'lily' that 'goodman' has selected units' abilities(2:10, 2:10, 3:15, 3:15, 1:2).
SetUnitsAbilities: 'goodman' has selected units' abilities(2:10, 2:10, 3:15, 3:15, 1:2).
SetUnitsAbilities: notify 'goodman' that 'lily' has selected units' abilities(1:1, 2:8, 3:13, 2:8, 1:1).
SetUnitsAbilities: 'lily' has selected units' abilities(1:1, 2:8, 3:13, 2:8, 1:1).
GameAction: notify 'goodman' that 'lily' has move unit(3) to (1, 3), perform ability(2).
GameAction: 'lily' moves unit(3) to (1, 3), perform ability(2).
GameMove: notify 'lily' that 'goodman' has move unit(4) to (4, 2).
GameMove: 'goodman' moves unit(4) to (4, 2).
QuitPlay: notify 'goodman' that 'lily' has quit battle.
QuitPlay: lily quit battle.
Logout: lily has logged off.
Session 99d791d9-e4a0-40cb-839a-01b665f04b4d is ended.
Logout: goodman has logged off.
Session 10bc2992-eeb5-4cf1-9492-c06773d050d9 is ended.


C:\Windows\system32\cmd.exe
lily login.
goodman login.
quickplay matched, goodman's opponent is lily, lily will play first.
quickplay matched, lily's opponent is goodman, lily will play first.
goodman selects commander 2.
lily selects commander 1.
goodman's opponent selects commander 1.
lily's opponent selects commander 2.
lily selects units(1, 2, 3, 2, 1).
goodman selects units(2, 2, 3, 3, 1).
goodman's opponent selects units(1, 2, 3, 2, 1).
lily's opponent selects units(2, 2, 3, 3, 1).
lily selects units' abilities(1:1, 2:8, 3:13, 2:8, 1:1).
goodman selects units' abilities(2:10, 2:10, 3:15, 3:15, 1:2).
lily's action: result(True), info().
goodman's opponent selects units' ablities(1:1, 2:8, 3:13, 2:8, 1:1).
lily's opponent selects units' ablities(2:10, 2:10, 3:15, 3:15, 1:2).
goodman's opponent move unit(3) to (1, 3), action=(2).
goodman's action: result(True), info().
lily's opponent move unit(4) to (4, 2).
lily's action: result(True), info().
goodman's opponent quits battle.
WARN: connection off, on_message will exit.
WARN: connection off, on_message will exit.
请按任意键继续. . .
lily: gameAction(3, 1, 3, 2)
goodman: gameMove(4, 4, 2)
lily: quit

Server: Deployment

The server code was created by one of the other members of the Server Team. After the server code was completed. We had our integration member deploy the code to a remote hosting service. We went through several hosting services such as: Amazon Web Services, thecity.sfsu.edu, and Google Cloud Services. We eventually settled on using Google Cloud as our hosting service.

In order to deploy this code, we first had to understand how it worked and how to compile the code so that we may have a working program on a remote machine. We first realized that the server code had to be compiled in a way so that a '.jar' file was created. The code was compiled using JDK 8, and this made it so that the code is executable with only one file. The code ran without issue on a local machine. After this it was time to set up the hosting service.

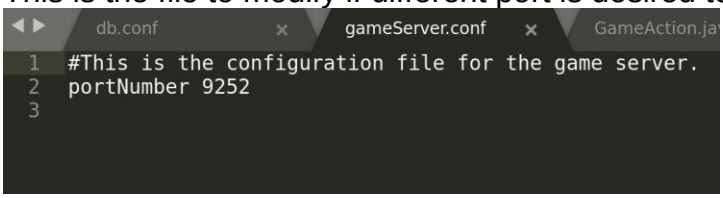
To deploy our server code. We first had to create a VM instance in Google Cloud Compute Engine. We chose to create a LAMP stack environment for our VM in order to use our server code on a remote network. The remote VM was then given a static IP address so that it may accessed whenever we wanted and so that our client code would keep persistence. To make sure that the server code connected to the database correctly, I had to modify the "db.conf" file in the server package with the correct credentials. In our case, we set the database to be on a localhost, which means that it lives in the same VM as the server.

In order to start the server, we just had to move to the correct directory of the server package, which contains the "start.jar" file, which is the executable that would run the server. To run this we had to type the command "java -jar start.jar" into the bash shell of the remote machine. With this our server was successfully deployed and ready for further testing and integration.

Instruction are as follow:

1. Change to directory of ffw.server.
2. Recurse strong copy to remote hosing service.
   Ex. "scp -r ffw.server 'username'@'remotehost':"
3. Log in remote host with ssh.
   Ex. "ssh 'username'@'remotehost'"
4. Change to directory of 'ffw.server'.
5. Change to ffw.server/conf and run "vim db.conf".
6. Modify db configurations as necessary.
7. Change to ffw.server root directory.
8. Run the 'start.jar' file using "java -jar start.jar"

This is the file to modify if different port is desired to be used as default:

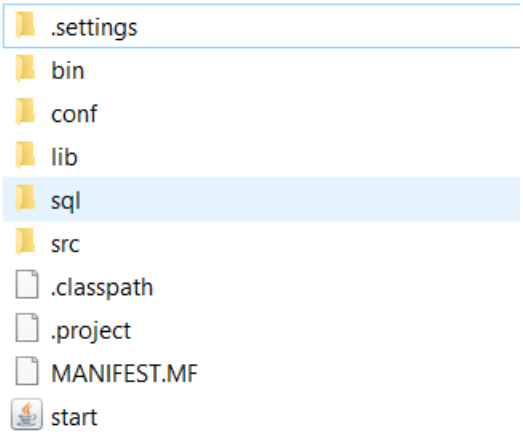


Database: Deployment

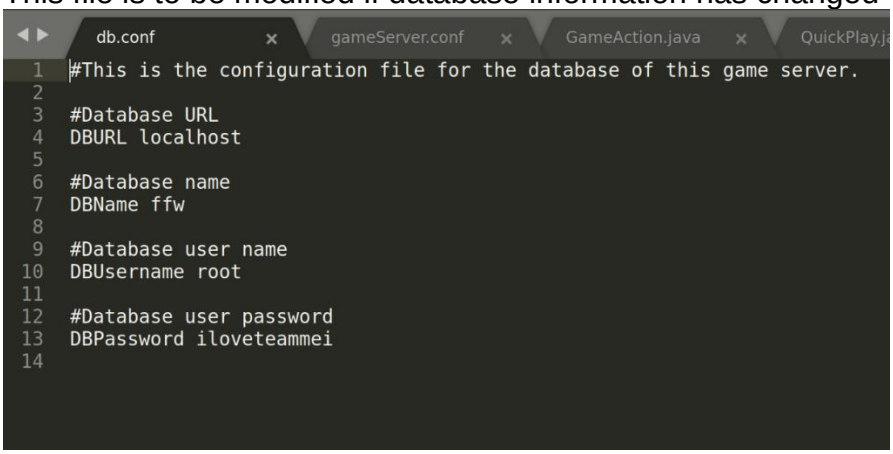
Our database went through a number of different iterations as far as the design. Our current design was handled by the Server and the Database teams. This database schema was then delivered to me along with the server code so that I may deploy it to a remote hosting service.

In order to do this we first had to set up a LAMP stack environment on a remote virtual machine on the Google Cloud Compute Engine. This same environment also hosted the server code which we would be using for our project. We chose to use MySQL as the database language for our database, and a MySQL server was then installed on the VM so that it may be used for our database.

To create the database on this VM was fairly simple since we were already given the dump file along with the server code. We ran this sql dump file and it created all the necessary tables for our game. This new database was then linked to the server code by manipulating the “db.conf” file to point to the correct database.



This file is to be modified if database information has changed



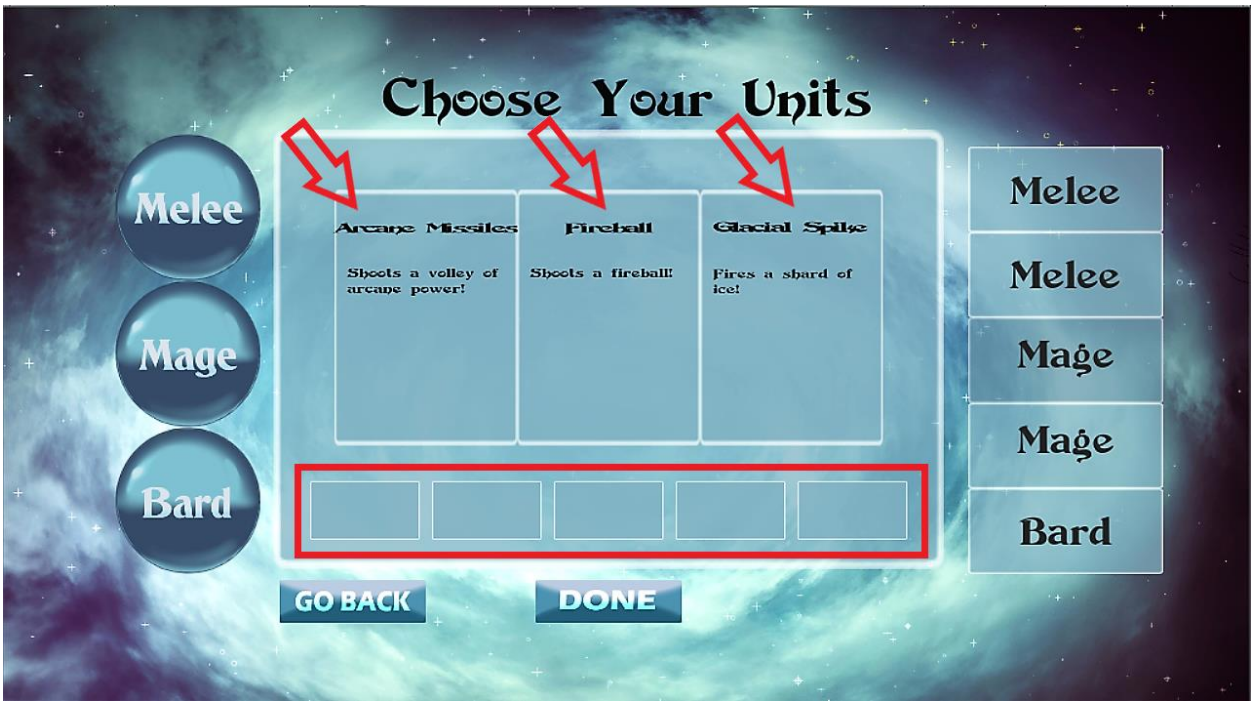
Server-Client Integration: Obstacles

Original Idea:

The loadout allowed players to customize their unit selection for their squad and abilities for each individual unit. Each unit was allowed 3 abilities chosen from a pool of 5 abilities placed in the bottom section of the scene.

Modified:

The way the server took the abilities and units from the client, only one ability per unit was taken into the database. In order to accommodate, we remove the ability to customize the abilities for each unit and preset 3 fixed abilities to every class and removed the ability selection in the bottom red box.



As shown below, the server method only takes in 2 arrays:

1. Long[ ] units where the array has 5 numbers; 0 = melee,1=mage, or 2=bard
2. Long[ ] abilities where the array has 5 numbers ranging from 0 to 4, each indicating an ability

Because only 2 arrays are being taken, one for a set of abilities each tied to one unit, the database is only receiving one of the 3 abilities we had originally wanted to do when designing the game.

```
public void writeToFile()
{
    File.Delete("PlayerData.txt");
    StreamWriter fileOut = new StreamWriter("PlayerData.txt");
    fileOut.Flush();
    LoadOutInfo[] info = StaticInfo.infoForDataBase;

    for (int i = 0; i < info.Length; i++)
    {
        fileOut.WriteLine(info[i].classSelected.ToString() + " " + info[i].skills[0].ToString() + " " + info[i].skills[1].ToString() + " " + info[i].skills[2].ToString());
        unitId[i] = info[i].classSelected + 1;
        abilityId[i] = 0;
    }
    fileOut.Close();
}
```

```
public result setUnitsAbilities(long[] units, long[] abilities)
{
    if (!tc.isConnected())
    {
        start();
        if (!tc.isConnected()) return new result(false, "connection lost");
    }

    tcp.request req = new tcp.request((short)cmd.REQ_SET_UNITS_ABILITIES);
    req.encode(units[0]);
    req.encode(abilities[0]);
    req.encode(units[1]);
    req.encode(abilities[1]);
    req.encode(units[2]);
    req.encode(abilities[2]);
    req.encode(units[3]);
    req.encode(abilities[3]);
    req.encode(units[4]);
    req.encode(abilities[4]);

    if (!send(req))
    {
        return new result(false, "network error: failed to send request");
    }
    return new result();
}
```

As you can see, unit [0] is set with ability [0] but no 2nd or 3rd ability. Therefore, we had to fix each class with 3 preset abilities that the player cannot change.

When we were able to connect two players to a match, we had to call the quickplay() method to connect them where the startBattle() method is called but playerID is never.

```
public void startBattle(Battle bat, int playerId, String who_first, String oppnent)
{
    this.battle = bat;
    this.playerId = playerId;

    putResponse(QuickPlay.doResponse(who_first, oppnent));
}
```

The way quickplay() and startBattle() is created, no queue/wait method was made in which it made it difficult to implement on the client side to start the game for the both players simultaneously. Meaning that when both players connect on the server, both players go into the battle scene at different times.

Player’s id also show’s up with no explanation and with no use here in this method.

```
53
54          new Thread()
55          {
56              public void run()
57              {
58                  while(!isDone)
59                  {
60                      if(players.size() >= 2)
61                      {
62                          GameClient c1 = null;
63                          GameClient c2 = null;
64
65                          synchronized(GameServer.this)
66                          {
67                              c1 = players.get(0);
68                              c2 = players.get(1);
69
70                              if(c1.isDone)
71                              {
72                                  players.remove(0);
73                                  continue;
74                              }
75
76                              if(c2.isDone)
77                              {
78                                  players.remove(1);
79                                  continue;
80                              }
81
82                              players.remove(0);
83                              players.remove(0);
84                          }
85
86                          Battle bat = new Battle(c1, c2);
87                          c1.startBattle(bat, 0, c1.user, c2.user);
88                          c2.startBattle(bat, 1, c1.user, c1.user);
```

This function needs to be modified to current client standards.

db.conf | gameServer.conf | GameAction.java | QuickPlay.java

```
1  package services;
2
3  import java.io.IOException;
4
5  import common.*;
6  import common.msg.*;
7  import main.Constants;
8  import main.GameClient;
9
10 class GameActionRsp extends GameResponse
11 {
12   public GameActionRsp(long who, int x, int y, long[] units/*, int[] damages*/)
13   {
14     buf = new MsgEncoder(Constants.RSP_OPPONENT_GAME_ACTION);
15     buf.addInt32(0);
16
17     buf.addLong(who);
18     buf.addInt32(x);
19     buf.addInt32(y);
20
21     buf.addLong(units/*[0]*/);
22     // buf.addInt32(damages[0]);
23
24     // buf.addLong(units[1]);
25     // buf.addInt32(damages[1]);
26
27     // buf.addLong(units[2]);
28     // buf.addInt32(damages[2]);
29
30     // buf.addLong(units[3]);
31     // buf.addInt32(damages[3]);
32
33     // buf.addLong(units[4]);
34     // buf.addInt32(damages[4]);
35   }
```

The database was made completely differently on the server than the original database we had originally created, due to miss communication it seems.

Original Idea:

ffw.sql | ffwmkii-db.sql

```
CREATE TABLE abilities (ability_id INT NOT NULL AUTO_INCREMENT,
                        description VARCHAR(255) NOT NULL,
                        unit_usable BOOLEAN,
                        ability_name VARCHAR(55) NOT NULL,
                        PRIMARY KEY(ability_id));

CREATE TABLE users (user_id INT NOT NULL AUTO_INCREMENT,
                    email VARCHAR(55),
                    username VARCHAR(55),
                    password TEXT,
                    PRIMARY KEY(user_id));

CREATE TABLE units (unit_id INT NOT NULL AUTO_INCREMENT,
                    unit_type VARCHAR(45),
                    hp INT NOT NULL,
                    PRIMARY KEY(unit_id));

CREATE TABLE user_units (user_unit_id INT NOT NULL,
                         unit_x INT,
                         unit_y INT,
                         PRIMARY KEY (user_unit_id),
                         FOREIGN KEY (unit_id) REFERENCES units(unit_id),
                         FOREIGN KEY (ability_id) REFERENCES abilities(ability_id),
                         FOREIGN KEY (user_id) REFERENCES users(user_id));

CREATE TABLE game (game_id INT NOT NULL,
                   PRIMARY KEY (game_id),
                   FOREIGN KEY (user_unit_id) REFERENCES user_units(user_unit_id));
```

Modified:

```
CREATE TABLE `t_commander` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(100) NOT NULL,
  `description` varchar(100) DEFAULT NULL,
  `image` varchar(300) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

/*Data for the table `t_commander` */

insert  into `t_commander`(`id`,`name`,`description`,`image`) values
(1,'commander1','commander 1',NULL),
(2,'commander2','commander 2',NULL);

/*Table structure for table `t_unit` */

DROP TABLE IF EXISTS `t_unit`;

CREATE TABLE `t_unit` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `label` varchar(100) NOT NULL,
  `type` varchar(100) NOT NULL,
  `description` varchar(200) DEFAULT NULL,
  `move` int(11) DEFAULT NULL,
  `basic_attack` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
```

```sql
DROP TABLE IF EXISTS `t_user`;

CREATE TABLE `t_user` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `email` varchar(100) NOT NULL,
  `name` varchar(100) NOT NULL,
  `passwd` varchar(100) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

/*Data for the table `t_user` */

insert  into `t_user`(`id`,`email`,`name`,`passwd`) values
(1,'lily@163.com','lily','123456'),
(2,'joe@gmail.com','joe','123456'),
(3,'goodman@hotmail.com','goodman','good'),
(4,'test@gmail.com','test','t');
```

This made it difficult to call correct items on the database we had created on gCloud database with the database functions on the server code. Meaning a lot of the methods went unused.

Conclusion and Future Suggestions

This team put many hours and late nights to make this game possible, and there is still a lot to do for this game to make it better and more fun to play than this initial version. A priority on our list was making sure we always had a working game in development while focusing on small tasks that could be easily changed to try out different ways pieces could be implemented. We have created a basic game upon which we believe others can come in, understand the components working in the game, and alter them to change how the game works to make it enjoyable. Below are some suggestions from the team of Spring 2018 on improvements to the game.

We would like to state that the asset store or premade code can make life a lot easier, spend time where it is needed. If someone else has a GOOD implementation of a component don't waste 60 hours of your time making it, just use their implementation if it is okay according to their license.

Pathfinding, hexagon tiles, unit movement, attacking, abilities, character models, animations, are all in a pretty solid state for the most part. The base map generator is also in rather good condition.

Abilities:
Make scriptable object, use new keyword for available abilities
--Have loadout grab ability information from the unit's available abilities
--Change on target ability to be able to have inherited classes run it as a baseline for damage or healing
--Maybe rework healing so that units changeHealth instead of separating into takeDamage and heal so that negative damage could just be handled as healing

Getters and setters could be removed to make code look more clean, with unity being very heavy on public components.

Audio was not handled the best for ability resources, some research should be done to improve this.

There may be alternative ways of doing abilities that are better, this is my first time working in unity so I am inexperienced.

Loadout:
Loadout should be scrapped completely and reworked. The visual and ability checks are fine, but the method of ability selection and the amount of non-automatically updated components makes it not maintainable. This class should automatically update on class/ability changes.

It is very cluttered and requires changes in 5 places to function on update.

Save as text file based on player name, to allow for more than one loadout (for local play)
Have a choice for unit design (there are 2 variations of each class) shown here

Unit Spawning:
Should be a self-contained class getting spawn positions based on player number (player 1 or player 2).

Should placement using raycasting in predesignated spawn zones of the map.
Reads the loadout file for the given player for units.

GameController / MouseController:
These are cluttered and very messy. A lot of the logic is handled in the update method of MouseController, which should be broken into subparts.

Look at this code. Try to understand what items do. There are methods that were used for unit placement and other irrelevant pieces of code as the unit placement did not work correctly. Not needed components should be deleted and this should have components broken down into clean tidy little methods.

Some of this logic is also flawed and not 100% working, as we did not have time to playtest everything.

On ability chosen it should highlight the effect radius of the ability, if the effect radius is 1 it should highlight all neighboring tiles on mouseover to display the area of effect.

Unit Selection:
There should be a method to select units (maybe with a right click) to look at unit information for the selected unit.

Ability Resource Paths:
These should be organized better, instead of just thrown into the resource folder.

GUI:
GUI and menus are very messy right now, need to be cleaned up and have logic organized neatly. When attacking the ability name of the attack should be displayed instead of 1, 2, 3.

There is GUI present within the files, it just is not implemented.

Unit Classes:
Should use inheritance properly instead of what it does now, or instead be one class only that is saved as prefabs for melee, bard, and mage to allow for easier manipulation.

Status effects and cooldowns could be added to this class.

For example poison could be handled by an integer that counts down every turn, same with ability cooldowns. Stun could also be added as an integer counter, cover, and many other components.

Visualization could be made for status effects as well.

```
public Unit(string charName, int maximumHealth, int currentHealth, int attack, int defense, int speed, int charNum)
{
    characterName = charName;
    maxHealth = maximumHealth;
    health = currentHealth;
    this.attack = attack;
    this.defense = defense;
    move = speed;
    charNumber = charNum;
    //Debug.Log("Calling " + characterName + " constructor and model number is " + charNumber);
}
```

Constructor should be changed to take abilities for the availableAbilities or a secondary constructor should be made.

Hexagon Prefab:
This was made in blender. I made it with only one material, it should have 2 different ones. One for the sides (allowing height to be more visible) and one for the top.

The base mesh is good for the purposes of this game, but the textures/ materials should be done better. Any image file can be used to reskin the tiles to make it better match the theme of the game.

Game end screen:
A game ends screen should be implemented, allowing for the user to return to the menus on game end

Map additions:
More maps should be made, these are easily done using my map generation class.

Menus:
Music is handled poorly here, should try to resolve this

Camera Controller:
Delete this monstrosity and make a camera that rotates about the center of the screen or by set degree amounts with maximums/ minimums instead of how fast the mouse is moved. Just import a premade camera if you can find one.

Resource Cleaning:
This code could use cleaning up of resources before coding starts/ removal of not used items

--There is likely more that should be changed, but this is a list of the components that stood out the most. The code also needs some love and cleaning up. There is obsolete code here and there.