Richard Robinson

Multiplayer Game Development
Individual Final Documentation

Table of Contents

0) Preamble

Email me at Richardjohnrobinson@gmail.com if you need help

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 and the other on the opposite side for an iteration I had, but for the presentation we moved spawning closer to be able to show abilities without walking across everything. The submitted version of the game has connections to a server, registration, log in, and a few other components done.

Not all abilities work as intended, or even work at all (may crash the game). The abilities that I implemented should not crash the game, but we did not have time for extensive playtesting and some game logic is handled sloppily or poorly. The working abilities will be listed in my personal work section: **3) Personal Work.**

The game should be playable at a basic level, allowing for units to be defeated, and a victory dance will be done for the winning player but there is no end game logic and it has not been sufficiently tested due to time constraints. There are some design choices that were poorly done, but these will be covered in the section: **4) Items to Change for Future Groups.** The goal of this documentation is yo explain design choices, how everything that I personally interacted with is done and how they can be improved or should have been done.

With this forward completed, I present to you:

1) Game Overview

Game Description:
First Fantasy Wars is a multiplayer game currently in development, made within the unity engine.

Game Set-Up
Each player starts with 5 units which can be chosen from 3 base classes: a ranged, a melee, and a support class. The ability choices are customizable, three out of five abilities are selectable for each class. These ability and class combinations can be saved as "Loadouts" so that they can be used immediately while playing instead of needing to reselect characters every game that is played.

Loadout:

Loadout Start Screen

--This is the starting loadout screen

--Units on right can be clicked on to see details/ swap out unit type

Ability Choice Screen

-For this screen the top “Mage” was chosen, these are the abilities

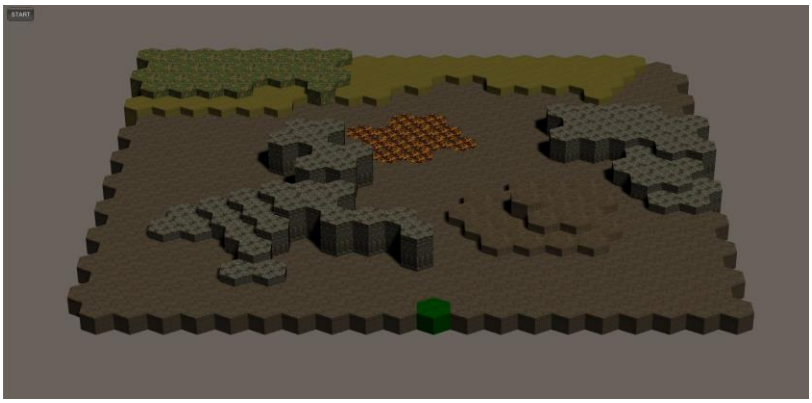
-Abilities can be chosen from below the descriptions

Change Unit Type

-The “Melee” was changed to a bard here

-This is done by clicking the Melee on the right then clicking the bard button on the left

-Abilities are empty by default on choices



Once done is clicked player is brought into the map

Gameplay

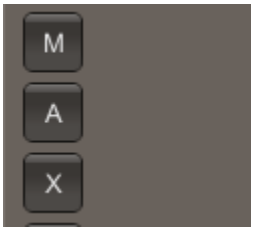
Click “Start” in top left corner

The turn sequence works as follows:

player one moves one unit then player two moves one unit and this 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.

The reasoning for alternating turns is so that one player cannot gain an unfair advantage by taking 5 turns in a row. Characters are alternated through to ensure more variety and that all units are forced to be used.

Each turn a unit can do the following actions: move, use an ability (or basic attack), or wait.



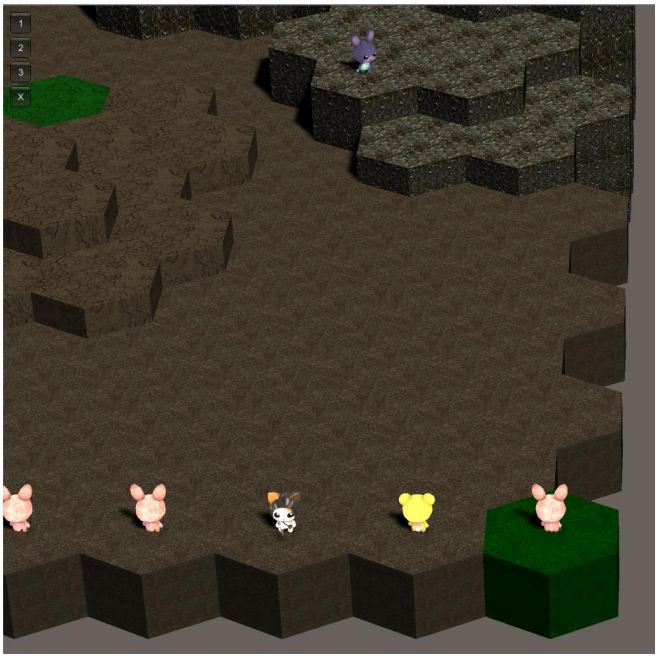
Actions are shown in top left corner

M = move
A = attack
X = cancel, or skip turn

M clicked

--Highlights movable tiles, mages have a movement range of 2

--This uses A* pathfinding for movement

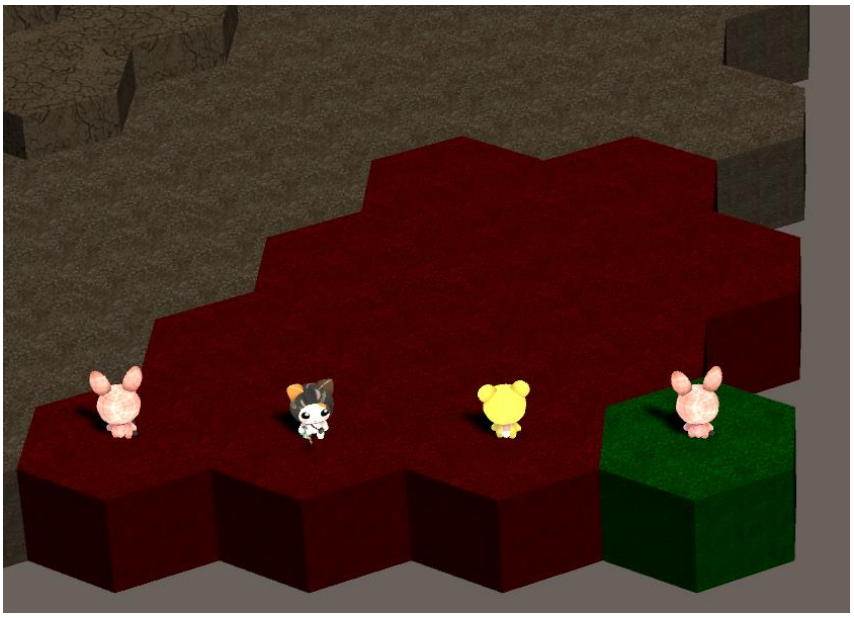


A clicked

--Ability choice is shown in top left corner

Select ability number to use.

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. This will be elaborated on in section 4)



Ability 1 was chosen

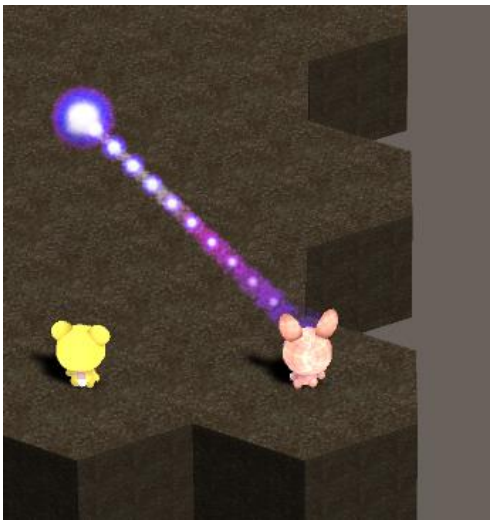
--Highlights attackable tiles

--Click on a tile to attack

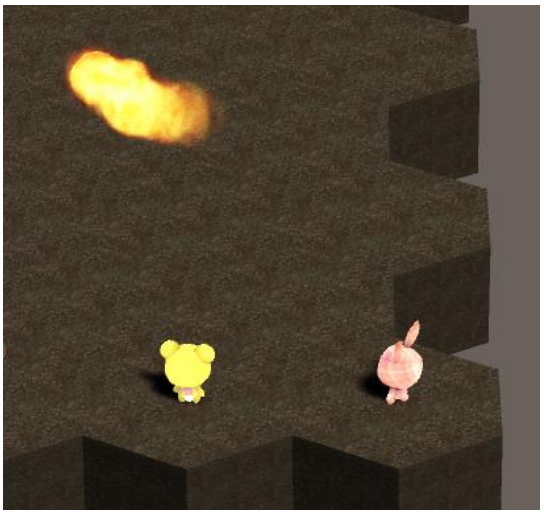
Below is a list of abilities currently implemented. They are not 100% complete in all ways they were supposed to work, but the baseline is there with animations and voiceovers/ sounds. Cooldowns are not currently used, but could be implemented fairly easily.

MAGE

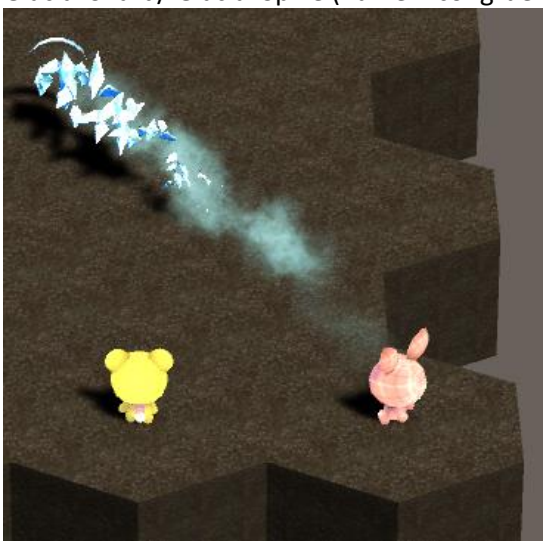
Arcane Missiles

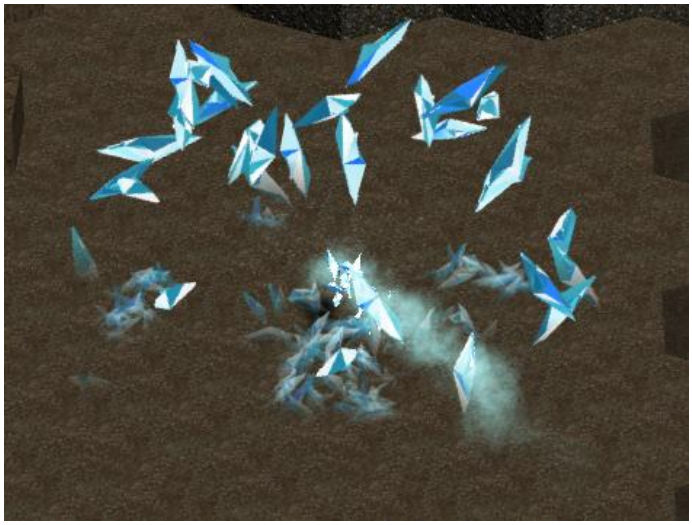


Fireball

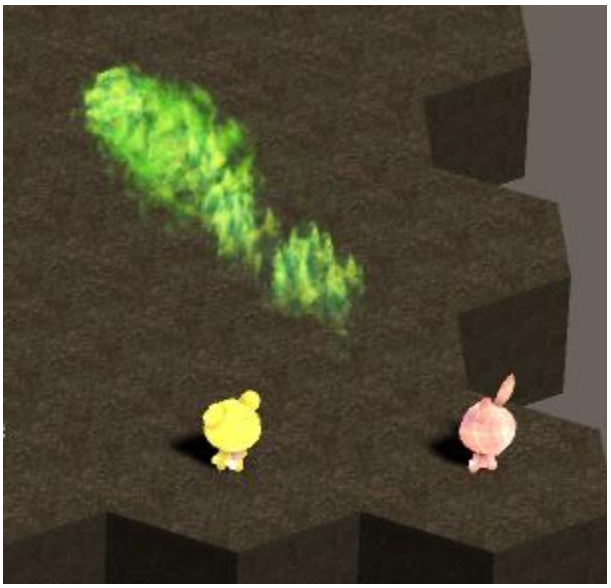


Glacial Shard/ Glacial Spike (name incongruency)





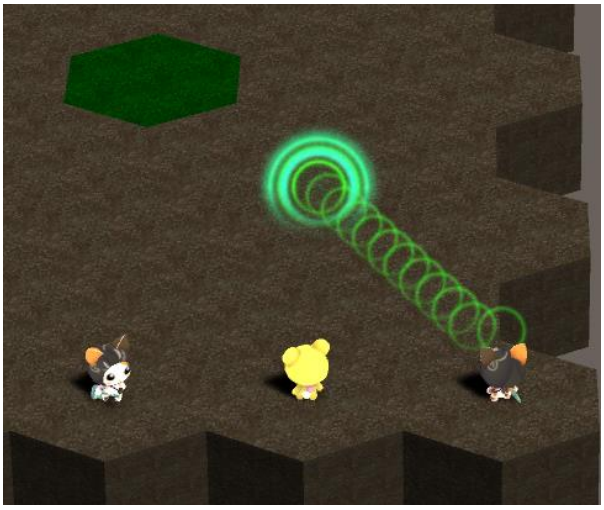
Poison

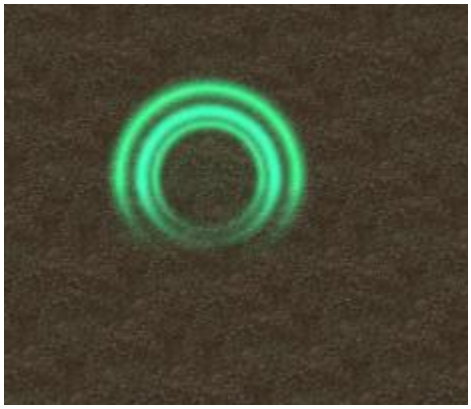


BARD

Restorative Serenade

Lament of Fate



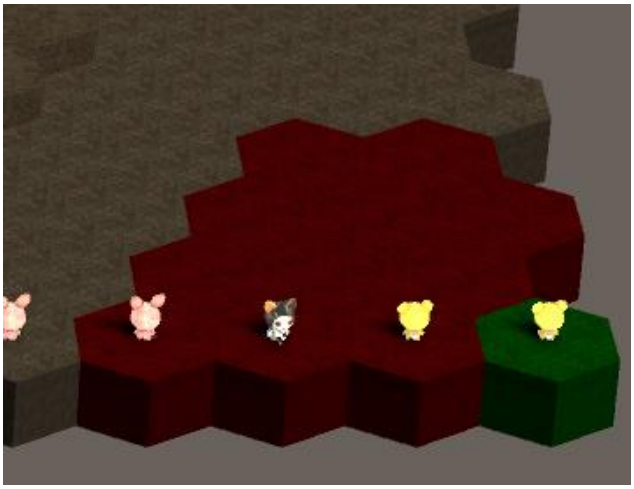


Viscous Mockery

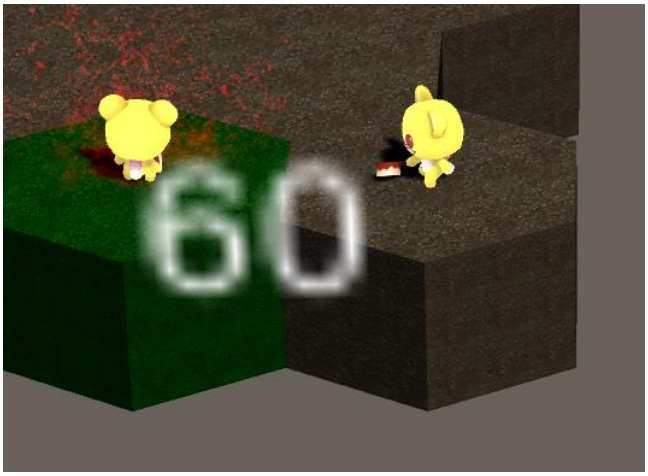
*This could be done using an array of strings to be randomly called for more insults/ taunts
*The display for this should be centered on middle of camera on the map not on the unit

Melee

Charge: DO NOT USE CHARGE ON A UNIT OR ON TILES THAT CANNOT BE REACHED BY WALKING, IT WILL LOCK UP (I did not make this ability, the checks for range are done incorrectly, it can only move to what would be walkable by that unit).

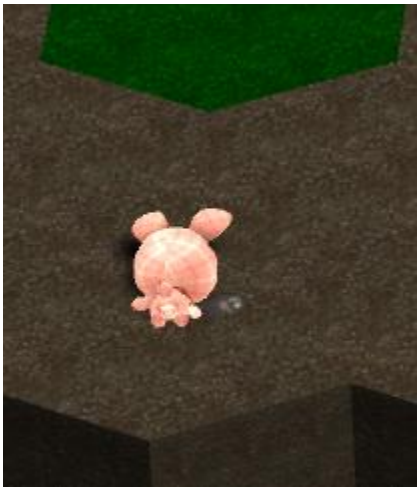


Brutal Blow



Poison Blades

Unit death



There are 2 possible death animations

Once all of one player's pieces are defeated victory or defeat is claimed and the game is finished. 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 a game is here.

2) Milestones

Details	Date
Concept Presentation	27 Feb 2018
Milestone 0: Game Concept Concept: Game concept details will be finalized and submitted.	1 Mar 2018
Milestone 1: Technical Design Client: Complete prototype designs/prefabs of maps, and non-textured character models. Art: Complete commander and unit designs. Server: Complete game server and test file Database: Complete ER Diagrams and schema for game. Audio: Complete Title Theme and menu SFX completed. Integration: Complete test server to be used for client Concept: Complete Milestone 1 Document & make changes to game logic if necessary	15 Mar 2018
Milestone 2: α Build Client: Complete prototype login, register, main menu, waiting, & loadout screens. As well as arena HUD. Art: Complete texture models for unit models and map tiles. Server: Complete multiple game session functionality and game chat. Database: Complete schemas & tables. Deploy to server. Audio: Complete Loadout screen theme and battle SFX. Integration: Link game server to client and database. Concept: Complete Milestone 2 Document & make changes to game logic if necessary.	27 Mar 2018
Milestone 3: *β* Build Client: Complete unit movement physics for combat. Complete unit interaction logic for moving through map. Complete unit animations. Art: Complete attack particle textures & animations. Server: Continue to test. Database: Complete modifications to database if necessary. Complete authentications of passwords and form submissions. Audio: Complete all necessary voice overs for battle sounds. Integration: Debug server and client interaction. Concept: Complete Milestone 3 Document & make changes to game logic if necessary.	17 Apr 2018

Milestone 4: Pre-Release Build Client: Complete unit turn list and unit battle logic. Art: Complete arena background(sky) models. Complete background models for main menu, loadout screens, & waiting screen. Server: Complete server maintenance & make changes as necessary. Database: Complete database maintenance & make changes as necessary. Audio: Complete multiple battle music themes. Integration: Merged Github branches and complete debugging of client and server. Concept: Complete Milestone 4 Document & make changes to game logic if necessary	24 Apr 2018
Milestone 5: Final Build Release Client: Complete unit pre-battle placement & debugging Art: Complete backgrounds for title screen & additional arena particles. Create poster art & trailer(optional). Server: Complete server maintenance & live deployment. Database: Complete database maintenance & make changes as necessary. Audio: Maintain audio assets. Create trailer music(optional) Integration: Merged Github branches and complete debugging. Concept: Complete Milestone 5 Document & make changes to game logic if necessary. Game presentation	8 May 2018

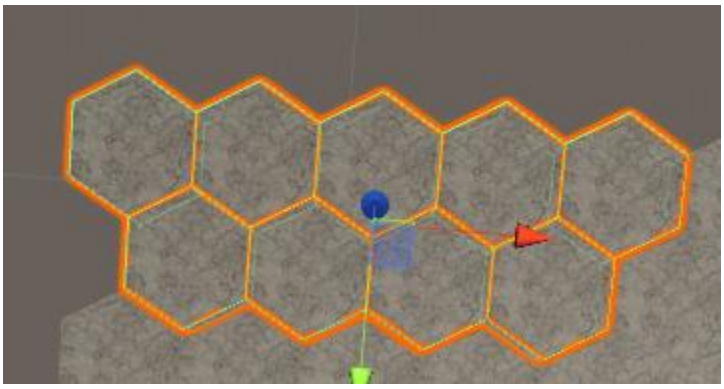
Personal Obligations For Milestones

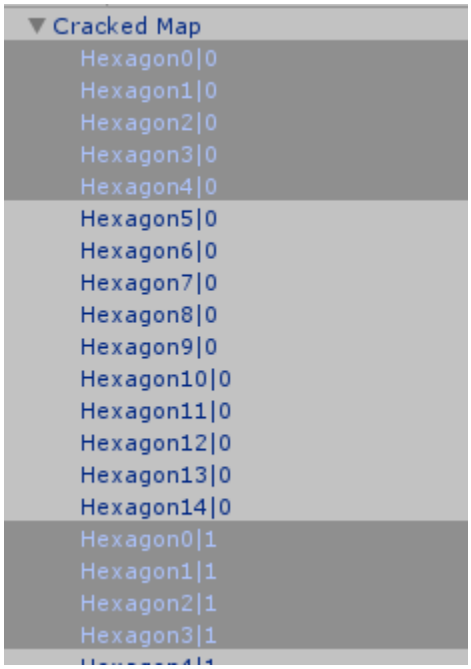
15-Mar	Complete Prototype Designs of Maps	YES
27-Mar	Menu Screens/ GUI	NO
27-Mar	Textures for Map Tiles	YES
17-Apr	Movement	YES
17-Apr	Unit animations	Mehi did
24-Apr	Turns	YES
24-Apr	Attacking	YES
8-May	Finish Debugging and Integration	NO

Integration was not completed. I helped connect clients to the server, register, and login but the online multiplayer component through online play is not currently usable.

3) Personal Work

Game World:
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





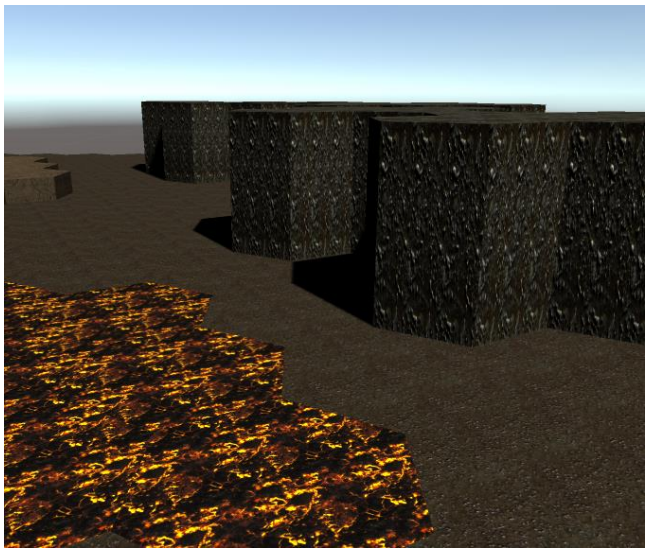
Left to right highlighted tiles:

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.

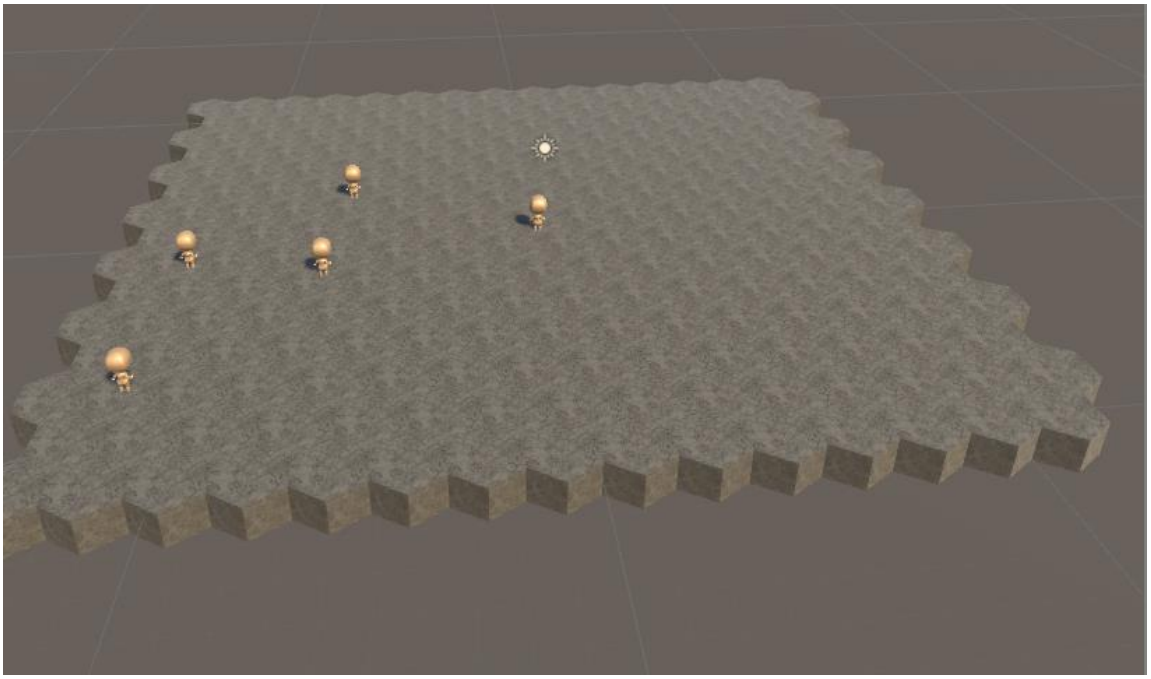
Units have X, Y coordinates as well: these correspond to the tile map coordinates.

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.

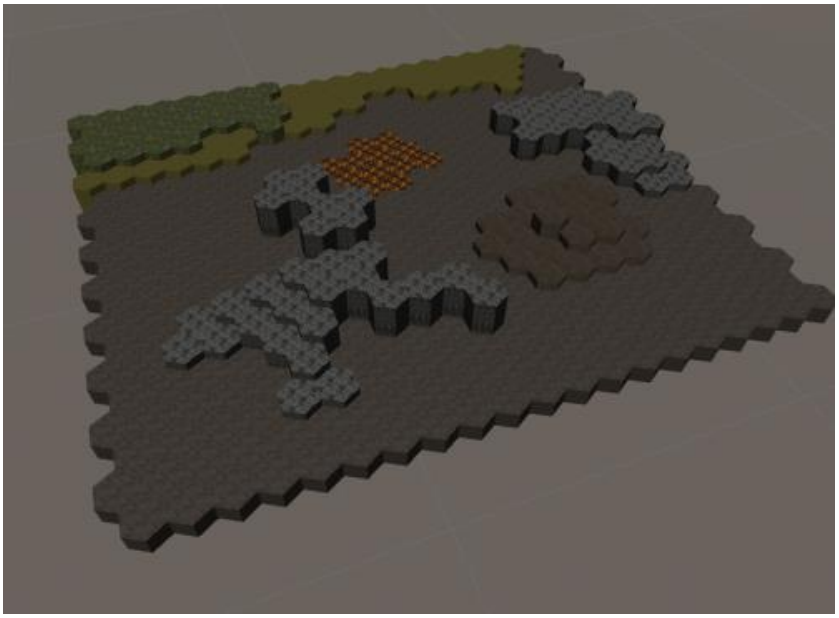


The movement takes into account height and whether tiles are able to walked on or not.

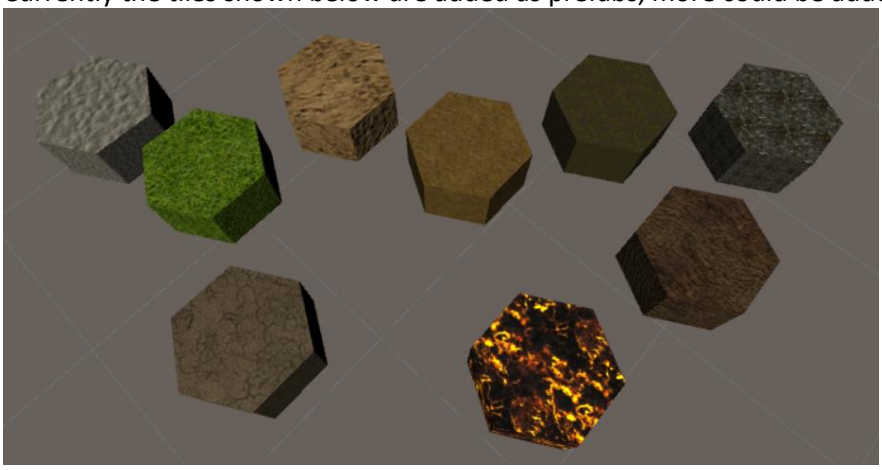
The basic foundation of the maps that I have worked on rely on a map generator script which lays tiles into a rectangle of set width and height. It places into rows and offsets the tiles by half of the tiles width on odd rows. This generates a simple map like the one shown below:



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.



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 (courtesy of Justin) 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.

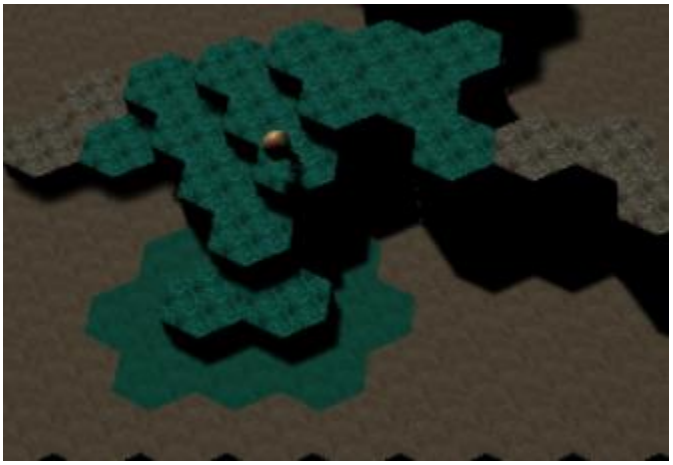
The A* algorithm uses distance of the current tile and the destination tile along the horizontal plane as it's heuristic function.

```
//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;
}
```

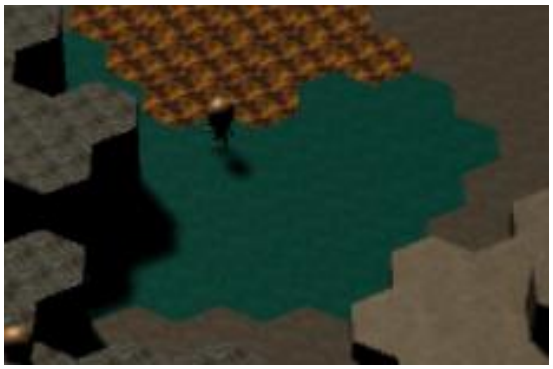
To return tiles within a certain range and maximum height difference the successor function is called until the desired node depth is reached (tracked within the node class), only grabbing successors that are within the maximum height difference constraint

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 I 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)
    {

    }
}
```

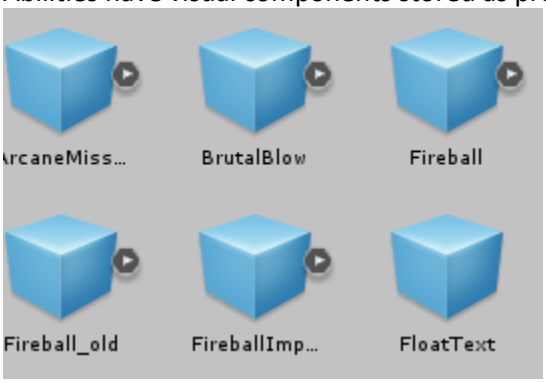
These classes should be refactored and have some attributes changed for future expansion, more detail in section 4) .

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

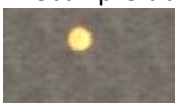
Example: fireball is one line of code

```
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:



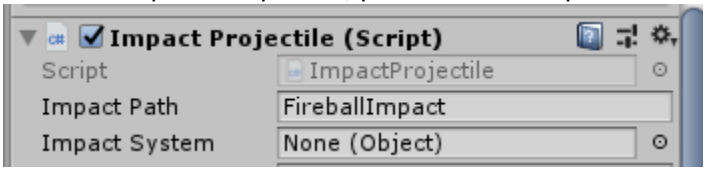
Fireball prefab:



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

The Code:
Projectiles:

Movement Code

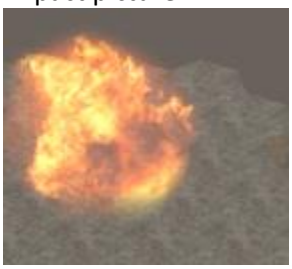
```
public bool Move(HexagonTile destinationTile) //Sets movement path
{
    this.destinationTile = destinationTile;
    this.movementPath = ActionController.projectilePathfinding(currentTile, destinationTile, 4);
    return true;
}
```

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())
    {
```

Audio Implementation:
Used FMOD, made life easy for audio implementation, but I never found out how to properly call triggers inline and spent a few hours trying to find out why songs were not looping.

Connection with server:
Used Dalu's code with namespaces removed to set up connection with client and server, log in, and registration. Further work was not done due to time constraints

4) Items to Change for Future Groups

First I 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. T[ry 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.

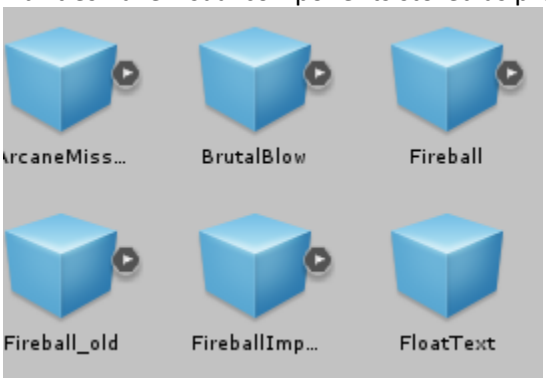
Resource Cleaning:
This code could use cleaning up of resources before coding starts/ removal of not used items

--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.

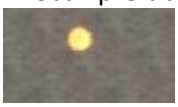
5) Help Using

Abilities:

Ability Visuals/ Ability Objects:
Abilities have visual components stored as prefabs:



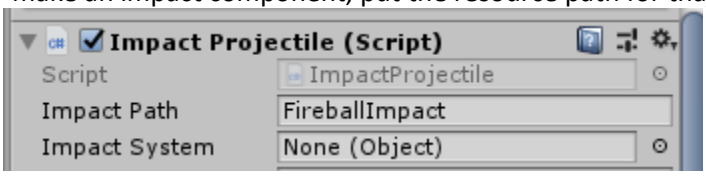
Fireball prefab:



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



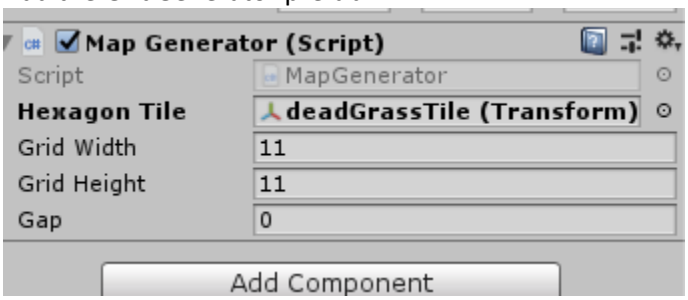
Impact component:
Get the visual
-Attach impact script to it
--it will just work

Save the items to the correct path

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

MAP GENERATION

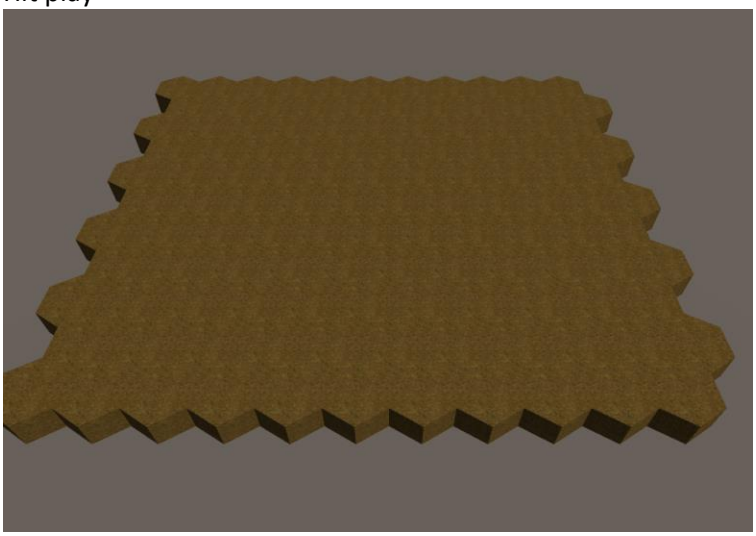
Add the GridGenerator prefab

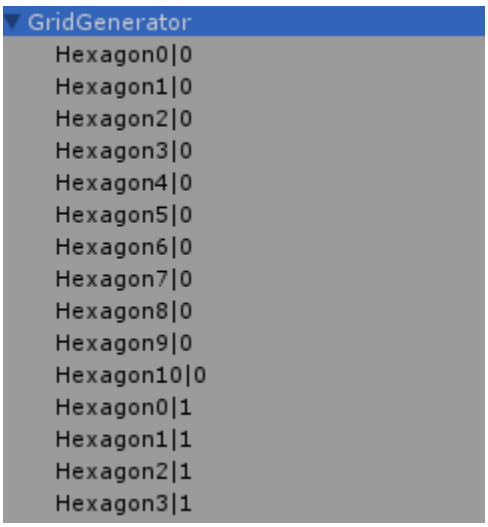


Drag a tile to the world

Attach that tile to the map generator script

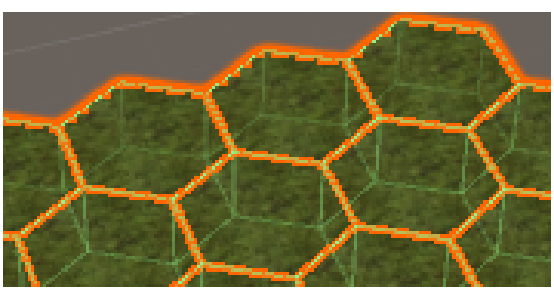
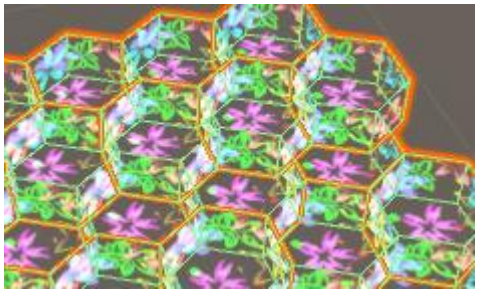
Hit play

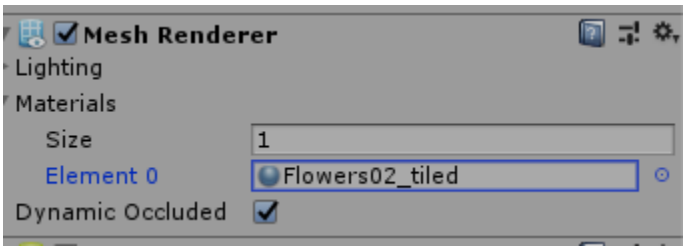




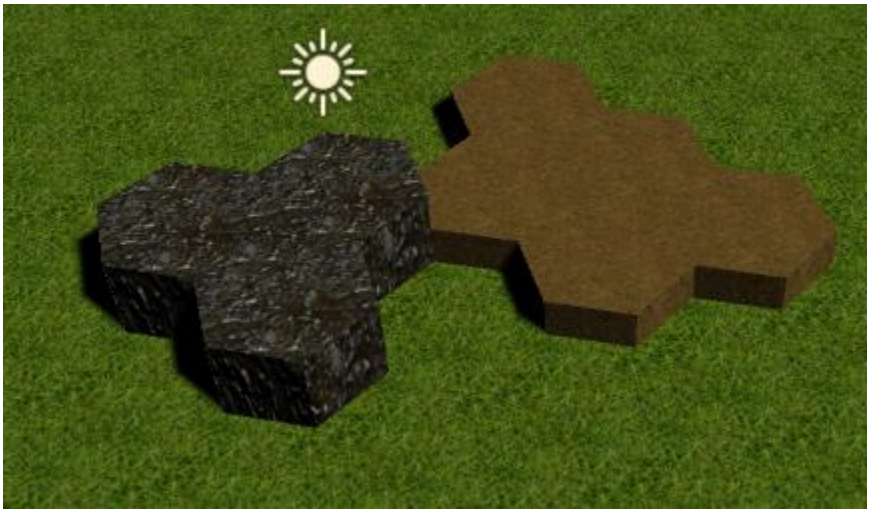
Rename gridGenerator to whatever you want to name it, remove the mapGenerator script

--Save it as a prefab (unity sometimes has texture issues when saving in play mode, just attach the texture to it afterwards if this is a problem, then it should be saved permanently)



Adjust height and textures OUTSIDE OF PLAY MODE



change z scale

change height

Loadout

I am not documenting how to use loadout because it is very confusing to use and I do not want to encourage its use.

It should at least be adapted to not need to take monobehavior objects and only extract information from the available abilities of the unit derived classes scripts

Adding abilities

```
public class Poison : ProjectileAbility
{
    public Poison() : base("Poison", 30, 4, "Poison", 3, "Deal 100% damage and target take damage over time", 0) {}
}
```

Make the resource as described above, pass the correct items into the constructor.

Additional effects can be added with overrides on activate.

These do not use radius currently, action controller has a method for getting tiles within a radius, just use that and look in the actionController methods for what you need.

On target ability:

Activate by default only places object

```
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;
}
```

If unit take damage method was changed this would not need abstraction, as the value could be passed as negative for healing.

```
public class RestorativeSeronade : OnTargetAbility
{
    public RestorativeSeronade() : base("RestorativeSeronade", 3, "RestorativeSeronade", 4,  "Heal another unit") {}



    public override bool activate(HexagonTile target, Unit castingUnit)
    {
        visualPrefab = Resources.Load(visualPath);
        GameObject projectile = Instantiate(visualPrefab) as GameObject;
        TargetedVisual targetVisual = projectile.GetComponent<TargetedVisual>();
        targetVisual.PlaceVisual(target);
        if (target.holdingUnit != null) target.holdingUnit.heal(40);
        return true;
    }
```

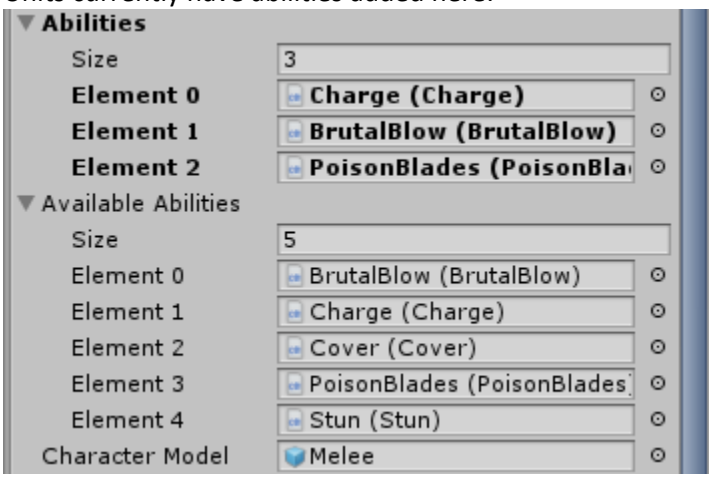
//Bard's restorative serenade heal.

Please change abilities to ScriptableObjects and use this

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

within classes for available abilities for auto updating and less ability clutter

Units currently have abilities added here:



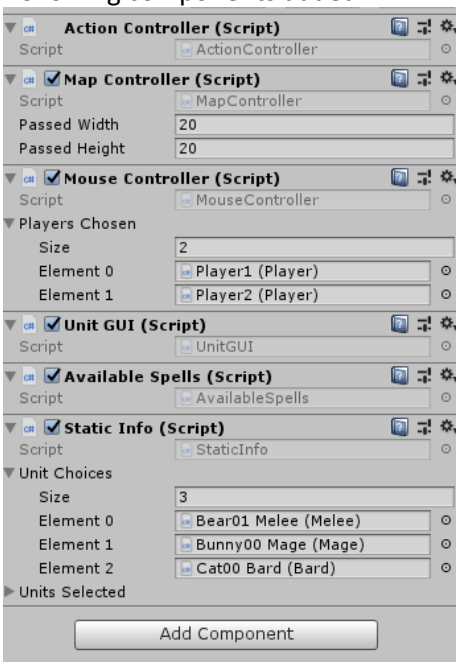
They can be drag and dropped if monobehavior.

If you change it to scriptable objects this never has to be changed and can be changed within the corresponding unit class.

Scene Requirements

-loadout set

-Following components added

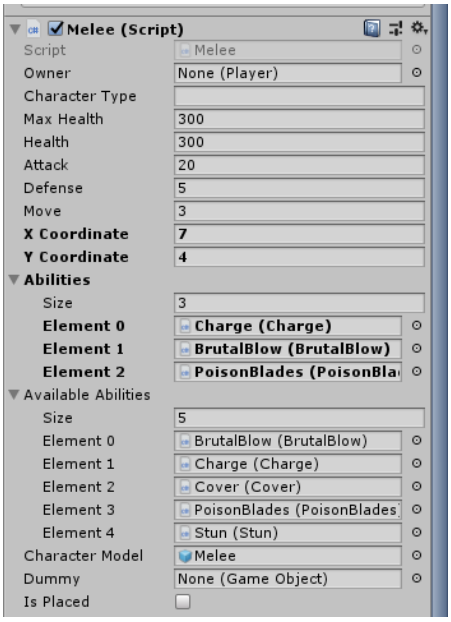


-Can preplace units or make new methods to add them

Preplacing:

Drag prefabs to game world, set 2 player objects, place units in it.

Class scripts go on prefabs.



add players to mouse controller

remove these lines in mouse controller if you are not using the default loadout

```
players[0] = new Player();
players[0].setArmy(GameObject.Find("GameControllers").GetComponent<StaticInfo>().LoadUnits());
players[0].setUnitsOnGameBoard();
```

If there are any issues please try to figure them out yourself if possible, but this should work (haven't tested with newest build, but unity is fairly good at pointing to what is wrong when it is visualized). Email me if any problems cannot be resolved.

6) Additional Details

Credit for the priority queue goes to BlueRaja's post from MIT
Spells and unit models are from the asset store in unity

After-thoughts:

This was my first exposure to Unity, I learned a great deal through this project as well as some things to not do. Overall my feelings for this project are mostly positive, however I wish there was more time to work on this project and a little less stress.

The base tutorials do little to prepare you for use of the fundamental base components of Unity and a lot of searching had to be done to find interactions and how to do specific things.

This project taught me about raycasting, linear interpolation, aided a bit in object-oriented design, exposed me to blender, audio, and visual creation.

It also got me over being stubborn and wanting to make everything from scratch. If there is a good resource out there already done implementing it saves huge headaches.

I am sure that some things got left out of this report, but I tried my best to be fairly thorough.

I hope whoever is reading this has a wonderful day.