Digital Dungeon

Design Document

Team 38

Andrew Batdorf  
Conner Isaacs  
Dillon Davis  
Vijay Viswan  
Zachary Williams

**Table of Contents**

1. Purpose…………………………………………….…….2
2. Design…………………………………………….……...4
3. Design Issues……………………………….…………..5
4. Design Details……………………………….…….…….9
5. Diagrams………………………………………………..12
6. UI Mockups………………………………………….….17

**Purpose**

Computer Science students at Purdue University currently do not have a fun way to learn about their department. The system we are designing is a procedurally generated dungeon-crawling game. Its purpose will be to inform its users (who may be potential students, or visitors at Purdue) of the many aspects of the Purdue Computer Science Department. The game will teach users the purpose of all of the core classes in computer science, give information on the Professors, as well as teach them about the different tracks a computer science student can focus in. The game will also teach potential students a very high-level understanding of basic algorithms/data structures, allowing them to become more accustomed to varying computer science topics.

**Functional Requirements**

1. Account  
   As a user,
2. I would like to be able to create an account.
3. I would like to be able to save game data to their account.
4. Character  
   As a user,
   1. I would like to be able to create a character.
   2. I would like to be able to customize their character’s appearance.
   3. I would like to be able to level their character and gain abilities.
   4. I would like to be able to equip items and gear in an inventory.
5. Tutorial  
   As a user,
   1. I would like to be shown how to control the game.
   2. I would like to be given an option to skip the tutorial.
6. Items  
   As a user,
   1. I would like to find and equip armor for my character.
   2. I would like to be able to upgrade my weapons.
7. Rooms  
   As a user,
   1. I would like to be able to explore different rooms.
   2. I would like to see a new map generated each time I play.
8. Quests  
   As a user,
   1. I would like to be able to partake in challenging NPC given quests.
   2. I would like to be able to solve puzzles.
9. Enemies  
   As a user,
   1. I would like to be able to combat enemies.
   2. I would like to be able to gain experience points and levels from enemies.
10. Companions  
    As a user,
    1. I would like to be able to recruit additional controllable companion characters.
    2. I would like to be able to give their companions gear.
11. Developer Console  
    As a developer,
    1. I would like to be able to access console to modify the environment.
    2. I would like to be able to transport to specific levels for debugging purposes.
    3. I would like to be able to able to activate various shortcuts through game progression.

**Non-functional Requirements**

1. Performance
   1. I would like the application to run smoothly at 30fps without crashing.
   2. I would like to be able to load the game within 5 seconds.
2. Server
   1. I would like the server/database to hold all the data collected/reviews sent in from users.
3. Appearance
   1. I would like key features, such as people or places, be recognizable from the CS department of Purdue.
4. Security
   1. I would like to protect a user’s password.
   2. I would like to be able to protect a user’s progress/saves in a game.
5. Usability

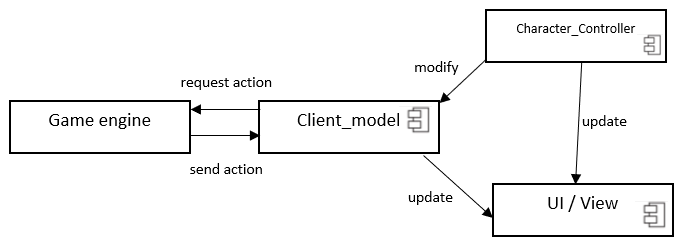
As a developer,

* 1. I would like the game to be intuitive to play.
  2. I would like the application to run on web browsers.

**Design Outline**

This project is a computer science themed, dungeon-crawler game. This game will use the client-server model, where the client is the player’s character and the server is the game level itself. The server will processes all actions proposed by the character, and respond with their success/fail status as well as interacting events in the map. The client will implement a model view controller (MVC) pattern, to manage presenting the UI to the user.

1. **Client** 
   * The user interacts with the game through a Unity based client.
   * The user interacts with the client through a modern web browser such as Google Chrome.
   * The client sends and receives game events from the server, by means of the GameManager class.
   * Actions proposed by the client may be denied by the server, to reject invalid actions by the player.
2. **Server**
   * The server receives events from the client, and tests their validity with respect to the current environment. If valid, the event is evaluated in the map environment, otherwise it is rejected.
   * The server also processes all other external events (enemy actions, environmental events, etc.) and reports their status after each *valid* action by the client.



**Design Issues**  
  
**Functional**

**Issue 1**: *How should character death be handled?*

* Option 1: Implement a system of save-points that allow the player to resume from play from earlier progress.
* Option 2: Develop combat and game progression around a permadeath system.

**Choice**: Option 2

**Discussion**: By developing a game around permadeath, the game will passively increase its difficult as the player progresses. This also serves to extend the replayability of the game since its not as trivial as to just keep pressing forward until progress is made. As an additional bonus, if the player experiences some unfortunate luck in terms of gear or weapon drops, by resetting on death the player is given a fresh new start at completing the game with potentially better items.

**Issue 2**: *How should the map be generated?*

* Option 1: Load one of several premade maps into the level.
* Option 2: Load several premade rooms into a level, rearrange them, and connect them with procedurally generated hallways.
* Option 3: Procedurally generate all room and hallways.

**Choice**: Option 2

**Discussion**: Sometimes entirely procedurally generated rooms can appear very unnatural or worse yet, break the game by producing unexpected shapes that other game logic didn't account for. However, some variation between maps is nice because it makes replaying a particular level feel like a new experience, and it also steadies the game progression by preventing the player from memorizing previous level maps.Therefore we are adopting a hybrid approach, where the rooms themselves are fixed, reliable structure, but their placement and hallways between them are randomly chosen in order to provide a degree of variation. This allows us to get the benefits of procedural generation, without the risks.

**Issue 3***: How should basic movement be implemented?*

* Option 1: Allow the player to move any distance in any direction, at any time.
* Option 2: Allow the player to move one step along a grid, at any time.
* Option 3: Allow the player to move one step along a grid, once per turn.

**Choice**: Option 3

**Discussion**: By locking the player to a grid, we can implement certain environmental interactions and combat actions that require precise positioning. This also means that the player can take as long as necessary to take action in combat, since movements are now turn-based. This puts emphasis on strategy, over just pure reaction-time.

**Issue 4**: *What factors should be considered for combat attacks?*

* Option 1: Just weapon damage.
* Option 2: Weapon damage and armor.
* Option 3: Weapon damage and accuracy.
* Option 4: Weapon damage, accuracy, armor, and evasion.

**Choice**: Option 4

**Discussion**: By considering four factors for each attack, we give the player several options for both offensive and defensive strategies. The player may decide to play it risky with high evasion armor in hopes of dodging most attacks, play it safe as a "tank" that can absorb most of the damage in armor, or some hybrid of the two. Similar logic can be applied to offense. This allows the player to develop their own play style, to better suit the amount of risk they are comfortable with. Additionally, this adds more variation and unpredictability to the attacks, making combat more engaging.

**Issue 5**: *How should the player move between levels?*

* Option 1: Automatically load the next level once the current level is completed.
* Option 2: Provide an exit at each level, which the player can enter to load the next level.
* Option 3: Provide a level selection menu, and unlock the next level for selection upon completion of the current level.

**Choice**: Option 2

**Discussion**: By providing an exit that the player can enter on their own gives them the option to explore the map further. In doing so, they discover collectibles, hidden "easter eggs", or simply overlooked drops that otherwise would have gone unnoticed. When the player feels prepared to enter the next level, they may do so by moving to the level exit. This approach also seems to be the least intrusive of the three- option 1 may force a loading screen and level progression upon the player that they didn't anticipate, while option 3 forces the player to temporarily exit gameplay to select a level.

**Non-functional**

**Issue 1**: *Which game engine should we use?*

* Option 1: Unreal game engine
* Option 2: Unity game engine

**Choice**: Option 2

**Discussion**: Unity is often compared to Unreal as being more intuitive and newbie-friendly, which means those inexperienced with either game engine will be able to learn it more quickly. Unity also has faster build times, so that means we can test our builds more frequently. Additionally, one of our group members has prior experience with Unity game development, which will likely help the rest of the team adjust more quickly.

**Issue 2**: *Which language is best suited for our game logic within Unity?*

* Option 1: C#
* Option 2: JavaScript

**Choice**: Option 1

**Discussion**: Most of our group members have a strong grasp on Java, which C# bears a strong resemblance to. This means there should be minimal effort in picking up the language; most of the time can be spent developing the game itself. Additionally, most documentation for Unity is written in C#, so time won't be wasted on trying to guess how to implement the same functionality in Option 2 (JavaScript).

**Issue 3**: *How should we store sprites for game objects?*

* Option 1: Store each image as its own .png file, in a /sprites directory
* Option 2: Create a single sprite sheet with all images in it

**Choice**: Option 1

**Discussion**: While a spritesheet reduces the number of files to manage and can be faster, it is also more difficult to implement, and is a bit of an overkill from a performance perspective. Most 2D games shouldn't have a performance issue in regards to sprite loading, so adding extra complexity just to reduce the amount image files is completely unnecessary. Subdirectories can be added to the /sprites directory to organize the images if necessary.

**Issue 4**: *How should we actually move entities?*

* Option 1: Use rigidbody containers and forces
* Option 2: Use a character controller and a custom script

**Choice**: Option 2

**Discussion**: Using a rigidbody could cause some issues with alignment to the grid, since we don’t get to specify exact positions but instead impulses for movement. Rigidbodies also have other quirks that arise from their innate interactions with the physics engine, so they may produce unexpected behavior whereas a character controller will allow us to specify exactly where we want everything to be at a given instant. Additionally, rigidbodies could be seen as an overkill for just simply movements, which add unnecessary complexity and latency- a character controller with a custom script will be far simpler, and even be faster since we can disable unused parts of the physics engine then.

**Issue 5**: *How should tiles and entities be drawn to the map?*

* Option 1: Use a single layer, and draw the sprites on top of eachother
* Option 2: Use multiple layers, and draw the layers on top of eachother

**Choice**: Option 2

**Discussion**: By using multiple layers, we can easily extract different parts of the map for individual viewing, which will be very useful for debugging. It also ensures that all entities will be drawn on top of the tiles, so no map updates (such as opening a door) will draw over entities, regardless of the order they are updated. As an additional bonus, layers that aren’t updated as often (the background for example) won’t be redrawn as frequently, which will give slightly better performance.

**Design Details**

**GameManager Class**:Class that is in charge of overall game functionality

* Level: An instance of the level class
* Player: A player object, that inherits from the character class

**Level Class**: Will contain variables for the overall levels (LevelNum, EnemyLevel, Tile, Room).

* Tile: Variable that will be a list of all the tile classes representing the tiles in the grid, all of which have their own properties
* LevelNum: An integer variable representing what level the player is currently on.
* EnemyLevel: An integer variable, which will be the max level that an enemy can be. The EnemyLevel variable will be a function of the LevelNum variable
* Room: A variable that will be a list of all the specialized rooms that have been spawned on this level (see different possible specialized rooms under Room class description).
* SpawnLocation(x, y): Determines the spawn location of the player on the grid (which is a 2d array). The Tile must be walkable (see Tile class description below)

**Tile Class**: Each tile on the overall grid (which is the map of the level) will be a Tile object. The tile class will have variables that represent different things that can happen on a tile (Walkable, Item, Position, PendingAttack, StatusEffects, Interactable, Hidden, Sprite)

* Walkable: Defines whether any character (player, enemy, companion) can walk on this tile or not
* Item: Defines what item is currently on this tile
* Position: The XY coordinate in the 2d array acting as the grid
* PendingAttack: This variable will be a state of sorts that defines that tile is about to be in the AoE of an attack. Any entity on this tile when the pendingAttack is switched off will take damage relative to the attack being casted
* StatusEffects
* Interactable: This variable will define if the player can interact with this tile or not
* Hidden: This variable defines if the player can see this tile or not
* Sprite: This variable says what sprite is on the tile (player, item, enemy, etc)
* enemyCount: Defines how many of an enemy are standing on the tile. This effects the attack speed of enemies on the tile.

**Room Class**:The room class has three variables (size(x, y), Tiles, and pos(x, y))

* Size(x, y): The size will be how many tiles large the room is. The variable x is the width of the room, while the variable y is the length of it.
* Tiles: A list of tile objects that all have individual properties (see above description of tile class)
* Pos(x, y): The pos variable will mark where on the level’s grid the room will spawn. The coordinates of this will mark the top-left most tile in the room.

**Rooms**

**All room classes inherit from room, and have a list of tile instances that all have their own properties (see description of tile class above)**

CompanionRoom Class

Shop Class

Computer Lab Class

Library Class

ClassroomAmbush Class

ServerRoom Class:

ElectrifiedWater Class

LockedInRoom Class

NeutralRoom Class

**Item Class**: The item class is the class that defines specifics about the different objects that can be picked up or dropped throughout the game. Each instance of item will have 3 variables describing its properties (sprite, equippable, consumable)

* Sprite: Defines which artwork to use for the object. Sword object will be given a sword sprite, enemy be given whatever artwork is chosen for enemies, etc
* Equippable: Defines if the instance of this item can be equipped by the player or not.
* Consumable: This variable defines if the object will be used by a countdown or not. For example, arrows will have a count that goes down on every use, or health potions will have a count that goes down when used.
* EquipPosition: Defines what position the item can be equipped in (if equippable is true). A helmet can only be equipped into the helmet slot, a sword into the weapon slot, etc.
* Value: an Integer variable describing how much gold the item is worth when the player sells to the shop. The shop will sell the item for 2x its value.

**Character Class**: Defines the instance of a character. A character can be the player, an enemy, or a companion, and will have three variables: Items, EquippedItems, Tile, State, character level

* Health: Integer defining the current health of the character.
* Items: A list of item objects that have been acquired/picked up by the player/enemy/companion. If an item object is in this list, it should be added to the players inventory, which can be done by using the items sprite attribute.
* EquippedItems: A list of item objects that the player has equipped. Same methodology as above.
* Tile: The current tile that the character is standing on. This allows the use of the tile’s sprite attribute to know to give it the character’s sprite and other functionalities (if there is an item on the tile, can be picked up, etc)
* State: The alert state of the character. There are three possible states: unalerted, alert, engaged. An Unalerted state signals that there is no threat to the current character. An alert state signals that there may be a threat nearby or in a line of sight. An engaged state means that the is in combat.
* Level: An integer variable to keep track of the character’s level
* Pos(x,y): Pair that says what the character’s current position in the grid is

**Player Class**: A class that inherits the variables from the character class, and has all attributes that will be unique to a player character. This includes items, equipped equipment, abilities, companion, and companionAbility

* Experience: A variable that holds the player’s current amount of experience
* Abilities: There are 4 ability slots and 7-8 abilities. This is a list of abilities that the player has currently chosen to be in his/her four slots.
* Companion: Defines the instance of a companion currently following the player.
* companionAbility: Defines what ability is set for your companions to use (1 ability slot)

**Enemy Class**: A class that inherits the variables from the character class, it defines instances of an enemy. All variables that needed for it come from the character class.

**Companion Class**:A class that inherits all variables from the character class, it defines

* Experience: A variable that holds the companion’s current amount of experience

**Ability Class**: A class that defines an ability and its uses. It has the variables that holds the properties of each ability. As follows: Shape, Damage, Cooldown, Sprite, Rank, MaxRank, DamageIncrease

* Shape: Defines what shape the AoE of the ability will take on the grid, and how big it is. For example, an ability may be a 5x5 square Area of Effect on the grid.
* BaseDamage: the damage that the ability starts out with at rank 1
* Damage: An integer variable that defines how much health will be subtracted from any characters within the ability’s Area of Effect. This integer is a function of the base damage + (Rank \* DamageIncrease)
* Cooldown: An integer variable that defines how long the player must wait between casts of this ability
* Sprite: Defines what artwork to use in the bottom left area of the User Interface (See Mockup of General UI)
* Rank: An integer variable that shows what rank the ability is currently at. (Players can level up abilities)
* MaxRank: An integer defining the maximum level that the instance of an ability can achieve
* DamageIncrease: The multiplier in damage that the ability will gain on each level up

**Weapon Class**: A class that inherits from the item class. They define the offensive combat stats for whoever is holding it

* Damage: How much damage the weapon does on attack
* Attack speed: How fast the weapon can attack
* Range: Integer defining how many tiles ahead the weapon can attack

**Armor Class**: A class that inherits from the item class, defining the defensive combat stats. These variables include reduction of damage in an enemy’s attacks and the chance of dodging.

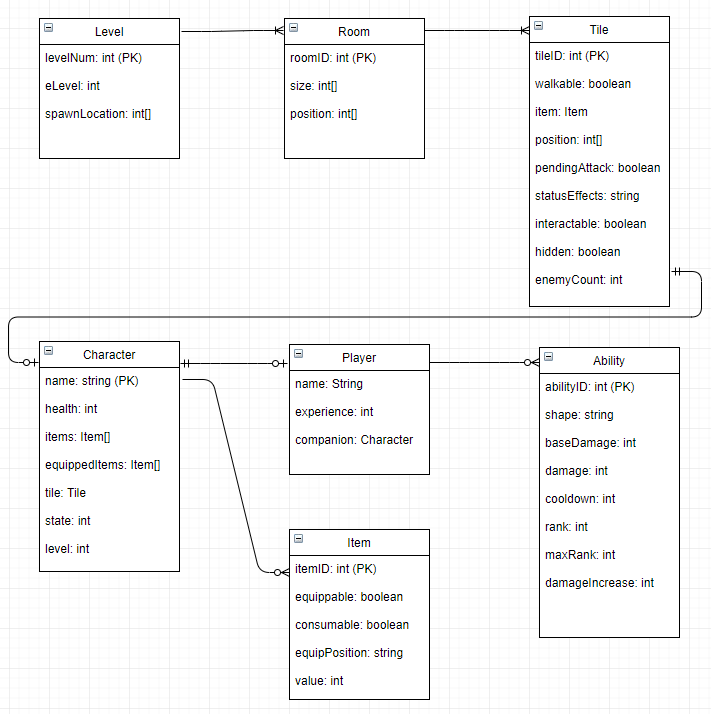
* DodgeChance: An integer variable defining how likely the character is to dodge any oncoming attacks
* MaxReduction: The maximum amount of damage that can be reduced for the character wearing it from an oncoming attack

**Data Class Level Diagram (next page):**

The level class has a room variable, that is an array of all of the specialized rooms that are generated on the level’s grid. Each room has a Tile variable that is an array of all Tiles that are a part of it. These Tile objects all have their own properties, such as if they’re walkable or not, and if they have items on them, etc. One of their properties is their position, which if it is the same as a character object’s position, we know the character is on that tile (which could have an item on it, etc), which allows us to know if the character can pick up an item or not, or other various actions.

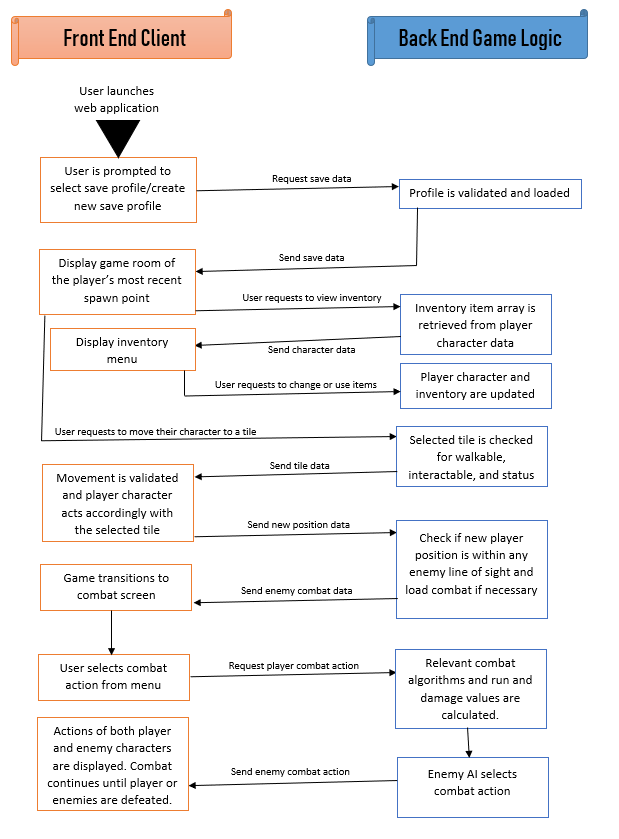
The character class has an item array, which is a list of all the items they have picked up, and they have properties that say whether they are equippable or not, and if they are consumable (which means their count goes down on every use)

The character class gets inherited by the player class, which has a variable storing all abilities that player has in their 4 ability slots.

****

**Activity Sequence Diagram (next page):**

This diagram shows the general flow of a user and their actions through our application. Upon launching our game through the user’s browser, they are prompted to a main menu with options to either create new save data for a new playthrough of the game, or select from a previously used save if one is available. Upon save selection, the game itself is loaded and displays the user’s most recent character room position according to the loaded save data. From here, they may check their character inventory menu, move their character, and interact with any nearby interactable tiles or enemies. All of these action requests are run through the game engine for validation and then the corresponding animations and changes to the game map are displayed to the user.

****

**Character State Diagram:**

This diagram shows the transition between states for characters. There are 3 states, unalerted, alert, and engaged.

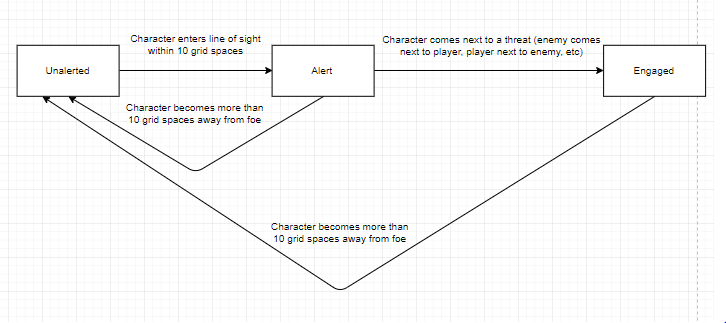
Unalerted will mean two things depending on if its an enemy or the player.

Enemy: The AI will walk around randomly

Player: The player may freely switch out abilities

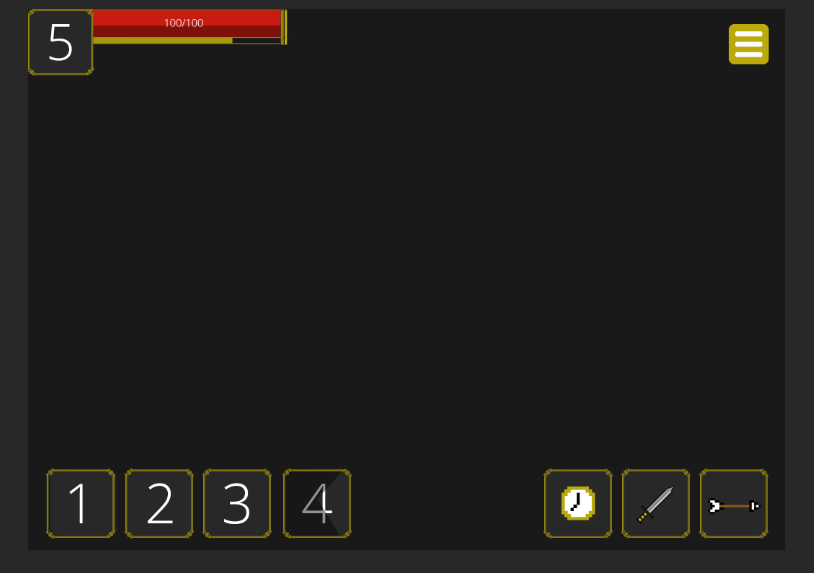
If an enemy and a player are within 10 spaces of each other and have line of sight, they will enter alert. This means the player cannot switch out abilities, and the enemy will start to move over to him.

When the player and enemy are next to each other, they enter the engaged state, which means that the enemy will begin attacking the player.

****

**UI Mockups**

**Gamplay UI Mockup:**

This mockup represents the UI that the player will see during normal gameplay. The player’s level, health, and experience will be on the top left of the screen. The menu button will be on the top right. The player’s ability choices will be the 4 buttons in the bottom left, and other options will be on the bottom right.****

**Inventory UI Mockup:**

This is a mockup of what the player will see when they open up their inventory. The block on the left will be for items that can be equipped (armor, weapons), the block on the right will hold any other items that the player has picked up or otherwise acquired.

****