Andrew Xu, Bill Wang

CC3K+ Design Document

Overview (describe the overall structure of your project)

For our implementation of ChamberCrawler3000+, we decided to follow the ECS (Entity-Component-System) software pattern. We represent the player, enemies, potions, and all items in the game as their own separate entity, and each of these entities will have their own set of components, corresponding to the information on each entity.

These entities are controlled by an EntityManager for each floor. The entity manager stores all entities on a floor in a vector. To facilitate interactions between our entities, (player attacks enemy, uses potion, etc) we use a specific system for each use case. Examples of these are the DisplaySystem, SpawnSystem, and CombatSystem, which handle displaying to the user, spawning enemies, and player combat respectively. Each of these systems has access to a specific floor's entity manager or all of the entity managers for all floors. All of these systems are also decoupled from one another, to have each system handle a separate part of game logic.

Our project folder is separated into four main folders src, include, build, and bin. Src is where we have our .cc files, include is where we have our .h files, build for the object files, and bin for the final output executable. This is derived from a general layout for relatively big C++ projects.

Within the include and src folders, we have subdirectories which organize our .cc and .h files. Components, entities, systems, etc. Having the .cc files all in one src folder makes it really easy to compile, and having all .h files in an include directory makes the including headers more intuitive. Note, components do not have a subdirectory in the src folder because they do not require implementation.

Updated UML

Differences in planned UML and current UML

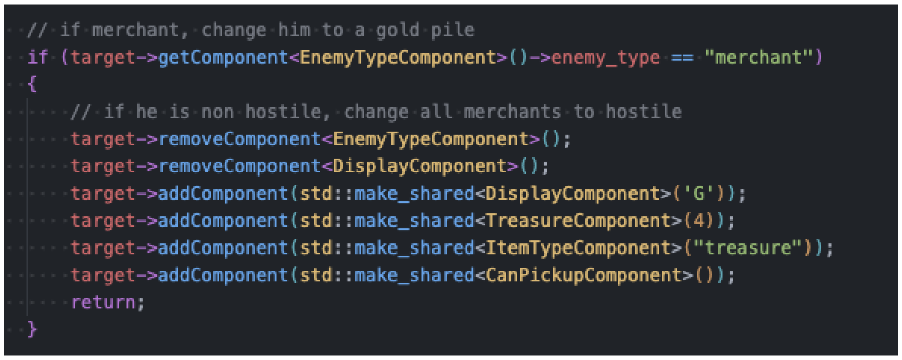
- Ability components no longer have methods. We decided to strictly follow the ECS pattern of having them represent data
    - For each ability, it serves as a “does this entity have this ability”
    - The functionality of the ability is in the system that uses it
        - IE: AllPositiveComponent -> In PotionSystem. Make all potions positive. On potion pickup, it changes the type to it’s positive counterpart

        - IE: LifeStealComponent -> In CombatSystem. Use damage to calculate how much life is stolen.
- We also added/removed some systems because either they required other systems to function (like EndGameSystem needing SpawnSystem if the player restarted), or it wasn't a big enough mechanic (like collect gold system which was just merged into CombatSystem).
- CompassCollectionSystem became a ItemSystem which facilitate all collection of items (including gold piles)
- Added and removed components
    - Enemy dropped gold got removed since we can just give enemy gold component and it will act as the same thing
    - Removed some fields in components if it was a boolean. This is because we can just check if the component exists, and it can serve as the boolean check
        - IE: CanPickUp had a pick-uppable boolean

<u>Design (describe the specific techniques you used to solve the various design challenges in the project)</u>

In the project specification, we noticed that there are many different types of players and enemies, each with their own special abilities. By following ECS, we were able to implement these special abilities very simply. We gave the entity a new ability component, and checked if that component existed on the entity in each of our systems. As an example, we noticed that the Dwarf and Orc races have a gold multiplier. If a user selected one of these races, then on creation of the player entity, there would be an extra "GoldMultiplierComponent" attached. Then in our systems where we handled adding gold to the user (item/combat), we checked for this component and correctly handled adding the right amount of gold.

One of the most interesting challenges we faced while implementing this project was spawning a merchant hoard on the death of a merchant. To give some background, our systems are designed to be decoupled from each other, so that they each handle a separate part of game logic. The death of a merchant is an event that is handled in the CombatSystem, while spawning any entity is handled in the SpawnSystem. We found it difficult to have these two systems communicate to each other, even though they are meant to be completely separate. To solve this, we realized that we did not need to spawn an entirely new entity when the merchant died. In ECS, entities are purely made up of their components, and switching up the components on the Merchant to convert it into a Merchant hoard was a simple solution.



```
// if merchant, change him to a gold pile
if (target->getComponent<EnemyTypeComponent>()->enemy_type == "merchant")
{
    // if he is non hostile, change all merchants to hostile
    target->removeComponent<EnemyTypeComponent>();
    target->removeComponent<DisplayComponent>();
    target->addComponent(std::make_shared<DisplayComponent>('G'));
    target->addComponent(std::make_shared<TreasureComponent>(4));
    target->addComponent(std::make_shared<ItemTypeComponent>("treasure"));
    target->addComponent(std::make_shared<CanPickupComponent>());
    return;
}
```

Thus we were able to handle this logic completely in the combat system, without needing any other system.

Both these problems are clear examples of how we used the flexibility of ECS to use simple logic to implement all required game mechanics.

Resilience to Change (describe how your design supports the possibility of various changes to the program specification)

ECS gives us many options for future changes to the program specification. As mentioned above, adding any abilities to a character could just be an extra ability.

Adding more complicated game logic is also easy, as we can create new systems to handle new game logic. Since each system is decoupled, each system can be developed without changes to existing systems. This means these systems can be independently developed, tested, and maintained.

Answers to Questions (the ones in your project specification)

How could you design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?

In our SpawnSystem, we have a spawnPlayer() method. This method gets passed in the desired race of the player and its desired coordinates. We then create the entity and assign the correct components according to the player's race.

With our design, it is very easy to add additional races. To achieve this, we can add another “if” block to our spawnPlayer() method, and make sure the created entity is getting the correct components.

How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

We handle it generally the same as generating a player. We call a similar spawnEnemy() method, which takes in the desired type of enemy instead of a player race. Enemies have different components than players, and they will get the correct components for their use.

For dragons specifically, we call a method spawnDragonAround() specifically after a dragon hoard or barrier suit is spawned. This function calls spawnEnemy() to spawn a dragon in a random coordinate around where the treasure/barrier suit was spawned.

How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

We can create a component for each of these special abilities. For example, GoldStealComponent, HealthRegenComponent, and HealthStealComponent. In our combat system we have an attack(*Entity& attacker, Entity& defender*) method that takes two entities, one for the attacker and defender. In this method we can add logic to handle each ability.

This idea extends to other systems as well. Health regeneration could be handled inside a new system called RegenSystem, which checks if entities have the HealthRegenComponent and regenerates their health.

What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

To implement temporary potions, we use the PotionEffectComponent to track the effects on the attack/defense stat for the player on any floor.

We use the AttackComponent and DefenseComponent to store the player’s base stats for attack and defense. When a player uses a temporary potion, the effect of that potion is added to the PlayerEffectComponent as an offset to the attack or defense stat.

In our combat system, we add the attack/defense base stat with its corresponding potion effect offset, which gives us the real stat to use in combat. The PotionEffectComponent offsets get cleared to 0 when the player enters a new floor or the game is reset. This lets us have to not explicitly track what potions a player has used.

How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hoards and the Barrier Suit?

Similar to generating the player races and different types of enemies, in our SpawnSystem we have a few methods for creating items. spawnPotion(), spawnTreasure(), and spawnItem(). We decided to split up these methods instead of having one big method for all items. This is because we noticed that potions, treasure, and other items such as compass/barrier suit spawn separately from one another, and being able to pass in what each method specifically needs such as potionType, and treasureValue would prove useful.

For spawning dragons around dragon hoards and barrier suits, we created a spawnDragonAround() method that selected a random spot around a dragon hoard/barrier suit to spawn a dragon in. This allowed us to reuse code when protecting the dragon hoards and the barrier suit.

Extra Credit Features (what you did, why they were challenging, how you solved them—if necessary)

We did not implement any extra features.

Final Questions (the last two questions in this document).

1. **What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?**

Andrew - For me, working on this project has taught me three main things. Firstly, clear and regular communication is crucial between group members. Being on the same page about project requirements, coding standards, and much more allows for much quicker development. Secondly, it's important for each team member to have defined roles and responsibilities. Luckily with our ECS design, each system was very decoupled which allowed us to clearly define what systems we needed to work on. This greatly reduced the time we needed to spend on minor issues, such as merge conflicts. FInally, I've learned that I work substantially better when pair programming with others working on the same project. Being able to constantly bounce off ideas and get instant feedback has been super helpful in the creation of this project.

Bill - One of the main things that I learned/reinforced is that communication is the key to success. Starting out, we had differing opinions on the architecture of the system. There were some issues agreeing on how to store the entities, the map, and data. Thankfully, we worked through that professionally and concluded that the ECS pattern is our best way to go. As we were developing, some ideas for new functions/systems came up and we discussed it thoroughly whether it's a good idea or not. This made the process a whole lot smoother, and more enjoyable.

Additionally, even with the help of version control like Git, combining our different implementations was more difficult than I thought. Surprisingly, it wasn't the merge conflicts that were the hardest issue, but the branches which got automatically merged into main. Sometimes, the merge added an extra return, or merged some other code which broke the flow of logic. Since it was merged automatically, the issue didn't arise until compiling and testing out features of another system. This taught me to look through the merge/commits more deeply.

Finally, choosing who you're partnered with matters. This privilege may not always be available in projects, but in this case it was. I can't imagine keeping my sanity if I worked by myself, or with a bad partner. Having a good partner doesn't just mean enjoying working with them, but also being able to talk to each other about different ideas and being open to criticism. I believe that our ability to work things out and provide feedback to each other made this project possible.

2. **What would you have done differently if you had the chance to start over?**

Andrew - If I had the chance to start this project over, I would spend more time writing documentation for our code and also implement better testing and error handling. Since we were working on separate systems, our work was beginning to become siloed, and writing more documentation would have helped alleviate that. Implementing better testing and error handling also would have provided a huge benefit, considering the amount of segmentation faults that we dealt with.

Bill - I would have definitely tried to keep a standard to my code. There were many times I switched between using std and not, which doesn't look very clean. I believe our code also needs some refactoring to try to reduce code repetition. Additionally, adding test cases or having a test suite would be beneficial as to make sure new implementation didn't break old implementation. If it did, it probably isn't decoupled right. Lastly, I believe we should've taken more time to figure out the "game system" and how the loop was going to work. I believe this can dramatically clean up our code to become more readable and robust.