Tourney Journey - Testing and Evaluation

CS4303 - Video Games

Personal Testing and Game Development

As Tourney Journey was being developed, I was constantly testing changes as I implemented them. This was crucial to my strategy while developing the game: I would implement one feature at a time, then test it thoroughly by playing through different scenarios until everything was working the way I wanted them to. I used two main approaches to personal testing: Code review and playthroughs. By combining these two, I became relatively confident that my game functioned as I envisioned it in my head.

Code review involved making small changes to code, then starting the game and testing the effect the changes in code had on the gameplay as quickly as possible. This was used partially in the beginning, before the game was in a working state, in order to identify and correct bugs as the game was being developed in order to prevent undetected bugs from causing bigger problems later on. It was at this point that I implemented the gravity and time effects, which provided core gameplay mechanics I could build my game on. Gravity in particular took a lot of testing and tweaking before it worked in a way that felt intuitive.

I also split up my implementation in a series of steps, which meant that at the end of each step my game was always in a working state. I started with just loading a tile map and displaying it on screen, I then added the player and implemented movement. After this I altered the map so it became a torus, meaning that moving off the edge of the map results in the player appearing on the other side of the map.

Following this I implemented a camera which follows the player, which is able to seamlessly move across the edges of the torus without the player noticing. I first did this by creating an orb which followed the player around, then after thoroughly testing this worked, I proceeded to center the camera on the orb and made it invisible. Getting the camera to move across edges seamlessly took a lot of testing before it worked correctly.

I then implemented the HUD, which was fairly trivial, followed by items. The reason I did it in this order was that picking up items is instantaneously reflected on the HUD, meaning that doing it in this order made it easier to test.

Finally, I added in enemies. As it turns out, enemies have a lot in common with the player, allowing me to reuse a lot of code I had previously written. As with the player, making sure gravity worked as expected on enemies took a lot of effort. The Monster AI also required a considerable amount of testing, and now works in a way which I am happy with. After this was done, the only thing remaining was to implement a system of waves and levels, as well as a starting screen, a game over screen and a winning screen, after which I felt my game engine was complete.

This is where playthrough testing came in. Whereas code review involved making small changes in the code and testing the effects as fast as possible, playthrough testing involved playing through the game without looking for anything in particular and noting things which needed to be improved as they occurred. This allowed me to find glitches and bugs in the gameplay which I would not have found my simply looking through the code. For example, One bug caused monsters to generate multiple gravestones when they were killed by damaging tiles, resulting in an arrayOutOfBoundsException. This was found during a playthrough, and I was not looking for it in particular when I discovered it. This was fixed by checking that the enemy's HP is over zero before killing it, preventing it from being killed multiple times.

Group Testing

I also sent multiple versions of my game to my group during various stages of development, and successively incorporated their feedback into my final game. This led to numerous features, such as the screen turning red as the player becomes low on HP, which was done as one of my group members found it hard to keep track of when the HP bar was becoming low. Another piece of feedback was that the game was too easy as the enemy AI would just jump in place as the player was above it, meaning that the player could kill enemies without having to press any keys. In response to this, I made the AI move in a random direction (left or right) after the player jumped on it, which effectively solved the problem. Another thing I found was that players found it hard to figure that they needed to jump on enemies to kill them. I solved this by hinting at it in the player manual, as well as designing the first wave of the game as a sort of tutorial, similar to what was done in the first Super Mario Bros.

I also received feedback from people outside my group as well, which led to features such as the blur effect which occurs when slowing down time. This originally occurred at an early stage in the game when I was experimenting with transparent tiles (this was before I had implemented backgrounds), but it was only showing a person in another group that I ended up deciding to implement it in the final product. I plan on uploading the game onto github at some point. The idea of health bars over monsters also came from another person.

Evaluation

I'm relatively happy with my finished product. I have designed a stable engine which is able to load players, enemies, and levels consisting of different tilesets from files, allowing campaigns to be made and sprites to be modified without making any changes to the code. I have also put considerable effort into creating sample levels which demonstrate various features and capabilities of the game. This being said, there are various things I would have implemented given more time. These are listed here roughly in order of decreasing importance.

- Create a level generator. The current method of making levels involves directly modifying level files. This ended up being a very tedious process, as I needed to restart the game every time I wanted to see changes I made to level files. A level editor would allow these changes to be shown in real time, and would remove the problem of needing to map between letters and their corresponding tiles. I feel that this could be implemented fairly easily due to the simple nature of the game.
- Create more levels. I originally had more levels planned, but did not have time to design them, as the process of creating levels took longer than I expected, as covered in the previous bullet point.
- Add music. I attempted to do this, but ended up not doing it for two reasons. Firstly, the lab machines which this game was developed on do not have any speakers. Secondly, When I attempted to implement it using Minim I found that the version of Java bundled with processing does not natively support Minix, and as a result I would have needed to manually copy over official java files into the processing libraries, which would have ruined the portability of my code as it would no longer work natively on the lab machines. Consecutively, I decided against this, although the code has been left commented out as it may be implemented in the future.
- Add animation to enemies and the players as well as more states. This was not implemented due to time constraints, but should be doable.
- The collision detection for large monsters is currently faulty, as collisions are only checked from corners, which sometimes results in false negatives when looking for collisions. Optimally every point along the border of a monster in increments of the tile size should be checked, which would prevent this from occurring.
- The jump height currently does not scale well as the size of monsters changes. I came up with two solutions to this, one of which is commented out.
- It may be possible to use linked lists instead of arrays in some places. One example would be when loading monsters or tiles, which would remove the need to specify the amount of monsters or tiles at the beginning of each file.

As with any game or large project, this is still a work in progress, as there will always be more features that could be implemented. Given the timeframe, however, I feel that Tourney Journey performs acceptably, plays smoothly, and the engine is flexible enough to make it easy to integrate assets into the game.