**Metallic Clashers Postmortem**

***The Idea*:** Create a match three combat game with a level progression within it. The game was designed to have different areas where the player can go and fight enemies using a match three system. The initial design allowed for items and special gems to be added in easily. The initial flow was set up easily enough, Brock would be the lead manager for the programmers and we had thought that Jonah would be the lead for the art side.

***Programmers*:** On the programming side we had little troubles. Most of our issues came from miscommunications that were quickly resolved within the group. Some issues we had were naming conventions, what styles and coding practices to use, and knowing exactly who was working on what.

For ***naming conventions***, this was one of those problems that came about when we started editing code. One of us would need information from another script and searching for that information was difficult at times. This problem was sorted out within the project with a naming convention for variables, protection level clarity, and function name conventions.

***Styling and practices*** are another problem we had as far as knowledge on the practices. Events and CoRoutines were things that we had heard about but only Ben really understood how to use them. Again, this was a problem that was addressed and Ben, who knew about them, taught the Brock and me how to use them.

Knowing who had what ***role*** was more of a vague idea. Each group member had their own sections of code that was written by them, such as one person doing the combat, one doing the stage selection and so forth. We never really said that, if there was a problem with combat, it was that person’s responsibility. If a problem arose, someone took care of it even if it may of not been their script. We did try to keep each person within their own coding area, but sometimes it came to a point where someone had to alter someone else’s code.

***Solution*:** Most of these conflicts could be solved with a meeting before the project begins explaining these conflicts beforehand. These problems are more of a management issue than anything else and could of been avoided had we all had the same level of skills. Either way, this is something that could've been discussed early on to prevent the problem, but we were just unaware.

***What went Well*:** In sight of the complications, the programmers communicated very well with one another. If complications arose that went beyond the bounds of one member, that member would seek help from a member that has handled where that problem arose from. If there was a major change to the code or to a mechanic, it would be discussed with the group. If there were complications with an asset, we would discuss what was wrong with it, figure a way to solve it, and bring the matter to one of the artist if the problem was too great for us to simply fix.

As a summary, the programmers did face conflicts and issues between code mainly, but it was solved through communication. Each group member helped by suggesting ideas to solve the problem and the group agreed on that solution.

***Team*:** The team in the later part of the project was able to come more together as a team, but not fully. The problems the team faced as a whole were: communications between the two sides, the pipeline between the artists to get assets to the programmers, and the time between assets.

***Communication*** between the programmers and the artists was made, but there was some loss information when speaking between the two sides. For example, if there was a problem with an UI asset, a programmer would go over and explain what exactly was wrong with the asset. The artist would accept this feedback, but when they would finish fixing the asset, some parts were still wrong. Communication was had between the two groups, but there was either a disconnect from what was said or information was not kept.

The communication issues also lead to a ***pipeline*** issue. Assets were always coming to the programmer side that required either changes from the artists, or massive reworks to code to make the art assets work properly. The pipeline for delivering assets from the artist to the programmers had to be reworked multiple times to accommodate for the errors within the assets. The pipeline was agreed to go from the artist, to their instructor, to the programming instructor, and then to the programmers. The farthest that that pipeline got was to the art instructor with him sending the assets directly to us, approving the assets with errors still within them. Reworking the pipeline did not help the workability of the assets, and we reverted to go back to the artists directly sending us stuff. From these errors and pipeline work, the project did suffer in art quality and some assets seemed like they did not go together.

Another issue sometimes was ***getting an asset*** all together. Brandon in particular has almost no assets in the game because his work has been redone and edited by Anh. The project only really has three models: the hallway and two enemies. All three of which done by Jonah, within the span of the project. Anh had done most of the UI assets such as the buttons, title, grid, node lines, UI boxes, and both gems. Brandon has done only the background of the stage selection screen and the nodes themselves. While the game has lots of assets, most of the ones used were all made by Anh, even if they did have some mistakes.

***Solutions*:** Even with the problems the team has faced, there are solutions to solve the problems mentioned above. To start, the team needed to agree, as a whole, on the art style and how to send assets to the programmers. With a pipeline firmly defined at the beginning and with a guideline on how to submit work, getting the assets in a timely manner is simple. Next, to fix the issues with the assets, there should of be some kind of standard to hold each asset to. Rules like proper whitespacing, layering, and size of the images should have all been defined beforehand to prevent any problems. We had believed that the artists had prior knowledge to all these kinds of issues but we were wrong.

***What went Well*:** With all of the conflicts out of the way, the team did do things well. When a mistake was made, the team took charge. When flaws were pointed out to the artists, they were quick to try and correct them. When the artists asked the programmers if the art can be displayed a certain way or have thing behave differently, the programmers worked to get the artist what they wanted. The team did produce what could be considered the demo of the game within a somewhat long time span.

***Myself*:** There were things that I personally had troubles with looking back on the project. Looking back I have solutions to improve upon but I still struggled with checking over work, keeping my code genetic to be used by other people, and making a decision that wasn’t mine to make.

While I try to write good code, I keep finding myself ***forgetting to test and check my work***. While my code is good to use, when I make a commit and pull request, I always forget about one piece of code that needed to be changed as well. One good example of this was the stamina system where I had forgotten to commit the commented lines to prevent it from saving after each time the application closes. This almost caused conflicts on the branches, but Ben was keen to see it and inform me about it before any conflict was made.

My attempts at ***generic coding*** have improved but still lacked in areas. This is where I think I struggled the most. I try to keep my code as simple as possible so that others can read it. My code does suffer at times from lack of using naming conventions and comments to explain the more complex parts of the code. Also, most of my code is designed to be used in a particular way or in a certain fashion to accomplish a certain task. There are a few areas where I could of reused code if I had programmed it to take in a wider range of inputs.

The previous two problems have been more with my code, but there were times I feel like I made a ***decision that was someone else’s*** choice. As a team, we had had meetings every so often. There are a few times I could recall that Ben did not show up for some of the meetings. Brock and myself deterred most of the decision making to Ben because he could see farther down the project than we could and had a better understanding of how things would work. On the times Ben was not here for a meeting, I would make decisions for the programmers based off of reason. I feel like Brock should of made that decision and not myself because he was our manager. I don’t know the reason why Brock allowed me to make the decisions, but he seemed ok with it at the time.

***Solutions*:** To start off with, I need to start checking my work more thoroughly before submitting it to be reviewed. Checking not only the code, but also the results. Questions to ask before submitting work is: “Were the values that were changed within acceptable range?”, “Was the values that were changed the only values to change as a result of my code?”, and “Does this code need to be separated from the the rest in its own script?”. For generic coding, I can employ a similar solution. Questions to ask about my generic coding can be: “Is there a similar piece of code somewhere else in the project?”, “Can the two code pieces be put together to accept a wider range of inputs?”, and “Can this piece of code be used elsewhere in the project?”. Asking these questions before work is submitted can boost how generic my code is. Lastly, I should consult with my group before I say that a decision is made. Asking other group members and leaders if it is alright to make a decision ensures that the group agrees with the decision. This would also allow other members to voice their input on each matter as well.

***What went Well*:** Ignoring all the issues I had throughout this project, I was able to accomplish work and understand how to contribute to the project. I was able to understand what each team member was doing and was able to explain to others what work I was doing. This helped when I would run into troubles coding and allowed those who were assisting me to have a full understanding of the problem. Also, I was able to quickly grasp new ideas and use them when it was necessary. When Ben had described what CoRoutines and Events were, I was able to effectively use them to accomplish my tasks. Finally, I was able to communicate ideas very plainly, so that our whole group could understand them. There was a moment where an artist asked if I could do something a certain way and was able to easily explain to him why I had to do it the way that I did.

***Contributions:***

* **GameManager** - saving game state and Scene transitioning.
* **Attributes** - initial creation and design
* **AudioManager** - playing music and sound effects for the game.
* **PlayerData** - designed to store and hold information about the player and saving it.
* **StaminaManager** - Designed to be a resource that came back over real time even if the game wasn’t being played.
* **CombatStatsBar** - used to display the important variables in relation to combat.
* **Enemy** - creation. This script is used to control the enemies of the game. It holds their statistics, how they take damage, and also controls when they attack.
* **EnemyManager** - this script controls where the enemies are placed in the scene, when they take damage, how they animate, and who is the current enemy.
* **TargetingEnemy** - initial design and early implementation.
* **MonoNode** - This script was to visually represent the nodes within the Stage Selection screen. Aslo contatined the Node class.
* **StageSelectionManager** - this class was to control how the nodes would be displayed, how they were to be statically set, handle loading in information about the world, set up the paths between the nodes, handling what happens when the scene ends, setting the parents for the nodes, handles dragging the nodes around, and displaying the stage info.
* **StaminaDisplay** - used to display stamina in text and visual format.
* **StageSelection Scene** - this is where all the stuff concerning selecting which area to go to is put together.