Team Game Project Postmortem

Overview

The team game project was a great experience. It went much smoother than the individual project and as such, was much more enjoyable. While we didn't achieve the full scope of what we set out to do, we nonetheless produced what I think is a great product. Our team worked very well together and I managed to learn several important lessons relating to working in team environments and working on game projects in general. The upfront planning of this game was much better than my individual game, however we still ran into several planning related issues during development and as such, hope to further improve this skill in the future. Overall I feel like I have improved as a game developer and programmer in general, and feel that my future productions will be much better as a result of this project.

What Went Right

We decided from the outset that we wanted to limit the scope of the game to give us the opportunity to deliver something well-polished. By the end of development, I felt that we had achieved that and that we had a great game that looked, felt and played well. While it didn't reach the full scope that we had aimed for, it nevertheless felt like a complete experience. With a little more work I expect that we could produce something even better.

Having now all had prior experience with our individual game projects, we were able to undertake a much more robust and informed planning process. I felt much more prepared going into development because we had some experience of what to expect. Having prior experience also meant that we could bring the best of our technical solutions from our previous individual games to the team project. For instance, I was able to apply my state machine solution to the game states and player states, Kadin bought a lot of his code for implementing data-driven development, Maya bought a lot of great manager classes such as SpriteManager and SoundManager and Robert bought his experience in creating animated sprites. This meant that the game started off with a much stronger foundation, was surprisingly modular, and involved much less mid-production refactoring. This in turn also allowed us to develop the game in a good order, setting up the foundations of our game before getting into the details of the gameplay. This gave us the ability to iterate our way through uncertain features without too much mess, confusion and refactoring. The manager classes also made memory management much easier and as such we experienced very few memory leaks throughout development.

During development we kept each other well informed of our commits and what we were working on. We tackled bugs, memory leaks and warnings as soon as they occurred and we kept our code fairly clean and well commented. We managed the game scope well, wrapping up development in time to allow for a significant amount of testing. This allowed us to discover and address gameplay bugs without too much last-minute panic.

Finally, I personally found that the team experience noticeably increased the quality of my work for several reasons. Firstly, I was forced to improve my code and take less shortcuts because I knew other people needed to understand and work with the code I produced. Secondly, being exposed to other people's code, technical solutions, and thought processes meant that I could adopt new and better ways of working. I'm now looking forward to working more within teams in the future,

whereas prior to the team game, my experience had been much the opposite, preferring to program alone and control the entirety of the work.

What Went Wrong

Despite this project running relatively smoothly, we did encounter some issues that set development back. One big issue was that at the beginning we underestimated the complexity and scale of the player class and it's functionality. Because of this, we decided that a player state machine design wasn't worth the upfront investment of time and energy and we chose to stick with an Enum driven player state system. This decision reflected poor planning and experience on our part, as once we were well into player development we realised that the system we had chosen was getting much more complex than we had anticipated. It wasn't until I was planning for the powerups that we realised that player states were going to get much more complicated and too hard to manage. At this point we decided to bite the bullet and transition to a state machine model for the player states before things got more out of control. This was absolutely the right decision, as our player class still had much more functionality to go and the confusing if statements and switches were already getting hard to interpret and taking up more and more time in debugging. The implementation of the player state machine ultimately greatly simplified the code and made it much easier to manage. However, refactoring our player class code to suit the player state machine still took significantly longer than it would have otherwise had we implemented it from the start. This experience was very similar to the one I had in my individual game when dealing with game states. Now that I have experienced this issue twice, I think I will be advocating for this approach from the beginning in my next project.

Also, in relation to the player state machine, I noticed that once implemented, the team didn't seem to fully understand it's purpose and functionality, for instance the clean-up and initialisation functions weren't being used correctly. This reflected poor communication, explanation, documentation and commenting on my part. This was partially because I was not yet confident in the solution, but also because I hadn't yet developed this essential work habit. This is something I definitely need to improve on in the future, making sure that my code is well explained and easy to understand and follow.

By the end of the development, we had unfortunately failed to reach the scope we wanted. We missed many features some of which were:

- downwards gameplay
- the shop
- a high score system
- music
- bouncing platforms etc.

We also missed out on the opportunity to add several details that would have bought our game more to life and made it more cohesive. For example, smoke animations on enemy deaths, particles around the player when they have powerups, background art that better suited the art style of the game. The list is long and we hope to implement these in the future. We may have gotten closer to reaching some of these additional features had we not had the player states issue and misspent some time on asset creation. Next time we will need to be more prepared and will need to better prioritise our time towards the most important development features and better invest time upfront where it is worthwhile.

This then leads to planning, which while being better than in my individual game, was still not perfect and it showed in the player states fiasco. Planning was even more important within the team environment because we couldn't always be across everything that everyone was doing, yet many features would be directly impacted by the way other team members were implementing their features. The best example of this was again the player states situation. Powerups were not appropriately planned in the TDD stage, so by the time I was actually planning the feature I realised that at least one of the powerups would be a player state in itself – whereas we had thought we had captured all possible player states already and hence chosen the Enum model. The Enum driven player state system was already getting very complicated and messy so adding to this was going to be complicated and meant that we had to totally redesign the player state. This is a great example of a costly issue that could have easily been planned for from the start. Now that we have more experience, I think we will do much better next time. Additionally, poor planning also meant that team autonomy wasn't as good as it could have been, as we needed to know how others were going to do something before we could do what we needed to do. A fully thought-out TDD prior to development would have gone a long way to remedy this problem also.

Finally, we experienced a notable and hard to solve bug at the very final stages of development. While we had been building in release mode for some time, we did add a couple of extra last-minute features to the user interface during that time. This ultimately caused some last minute panic and disappointment in our release build, which up until then had been fine. However, while disappointing, it was also a great lesson in the pitfalls of adding last minute features close to the gold milestone – and will yield another great lesson once we work out what the bug is.

Lessons Learnt

Speaking of lessons learned, there were many more gained throughout the process. Some pertained to technical skills and solutions, but personally I found that I learned the most about working in a team and good team practice. For instance, now that I was working in a team the importance of regular commits was much more salient to me and I found myself naturally doing it much more often. I also realised the importance of well-commented commits, well-commented and explained code, and good coding practice in general.

I also realised that working in a team offered many advantages that couldn't be gained in solo development. I found that I learned much more now that I was able to watch what my team members did, both in terms of good practice, but also in terms of their technical solutions. In a team you get to see the various different ways that others tend to solve problems. Another benefit to being in a team is that you don't have to worry about every single technical feature, and you can instead just focus on a handful of problems and create great solutions for them.

During development we also had a couple of pair and group programming sessions that I personally found very productive. I am convinced that this can be an extremely effective problem solving strategy. You can discuss and bounce ideas of each other, solutions tend to emerge based on simplicity and merit, you speak aloud and are constantly rubber ducking, and of course you have access to more than one problem solving tool set and set of experiences. It's also just more fun and turns coding into a more social experience. I will definitely be doing this more often in my programming career and consider it to be a game-changer.

Finally, I learned the importance of having a good game development infrastructure and tools from the get-go. A good game state machine, object managers and toolsets (e.g. .ini file setup) meant that development felt a lot more organised and modular. Many gameplay related problems were made much easier after these features were put in place. I realise that if we were to continue as a team in

the future, we would have even more good quality, tried and tested infrastructure that we could use, such as the new player states model and the animated sprites setup. This would both speed up development and offer more space and time for us to focus on more advanced, interesting and challenging game problems.

Conclusion

In conclusion, while the game didn't reach the scope we wanted, and while we encountered a few lessons along the way, I feel like the team game development project went very well. I experienced the benefits of working in a team, learned a lot about how best to operate within a team, and accrued some clear lessons in relation to planning and milestones. I had much more fun this time round than in the individual project and the resulting product was much better too. I feel like we produced a great product, worked together well as a team, and feel that any future productions would only get better with our combined experience, as well as with tools that we have now created and accrued.