LESSONS LEARNED

DEREK RANDALL
MAURO CHAVEZ
STEFAN DOUCETTE
JEFFREY PAULS

Slack and Zoom Communications: Text-based chat platforms such as Slack were very efficient for passing project messages to all team members across varying time zones. Furthermore, video chat platforms such as Zoom were helpful for clearly articulating weekly deliverable requirements and ensuring that all team members shared the same understanding of what was required. Without these video meetings,

Addressing Team Member Time Zone Differences: Although challenging, it is very possible to maintain a successful team that is spread across drastically different time zones. For real-time meetings, such as video conferencing, careful consideration must be taken in setting meeting times to ensure that all team members are able to attend. However, such teams will likely primarily rely on text-based chat platforms, such as Slack, that are not dependent on all team members being present simultaneously. Even though a majority of our team communications took place over Slack chat, we still found that we were able to coordinate and work effectively as a team. Without proper communication platforms, such a team would likely struggle.

Division of Labor: Every team member has their strengths and weaknesses. Some are better at coding, some are better at writing, and some are better at drawing/outlining. At the beginning of each week, we allowed our team members to decide what parts of the weekly deliverable they felt most comfortable working on. By giving them this freedom, each team member was able to apply their strengths and produce the best product possible. Furthermore, by breaking down massive deliverable requirements into their component parts, each individual only needed to research and work on a small number of details. Then, at the end of the week, we combined all finished components and team members reviewed the others' work in a Round-Robin review. This enabled us to make the most efficient use of each individual's work time while maximizing the quality of our products.

Working Ahead: Our team was very ambitious, and wanted to get to class design and coding solutions as soon as possible. Although tempting, jumping ahead to software design without establishing requirements and subsystem architecture can lead to constant architectural errors that must be corrected in later stages of the project. Instead, we recommend that future teams stick strictly to the requirements laid out in each week's deliverable. Furthermore, it helped to preview the next 1-2 weeks' deliverable requirements, to ensure the team was not doing work meant for future stages.

GUI Implementation/Prototyping: Throughout the Coding phase of our project, especially when creating the GUI, we quickly discovered that many seemingly easy features were actually far more difficult to implement or altered our game's state in unintended ways. For instance, our Design Document initially outlined that the Gameboard would control the game, but we discovered during Coding that it would have to be run by a Tkinter GUI boundary object. Although we had no way of knowing this previously, it likely would have helped to prototype

the GUI earlier (perhaps during the Minimal Increment) in order to discover implementation issues like this earlier in the project life cycle.

Development Tool Pre-Configuration and Standardization: Not all team members had the necessary software development tools installed and pre-configured prior to the Coding phase, and not all had extensive knowledge of GitHub operations. Therefore, some team members had trouble developing, pulling, and pushing code to the common Team GitHub. Therefore, we recommend that teams standardize early on what software development tools and packages all members should have. Also, it is recommended that teams conduct a dry-run on each of their computers to ensure they are all able to develop code and perform all required GitHub operations (pulls, pushing commits, etc.).

Framework Knowledge for Effective Planning and Design: Our design and planning documents were very accurate in capturing the implementation of our project until we had to utilize Tkinter (Python GUI module) to design user interfaces. This package is extremely quirky with API's that aren't intuitive or straightforward. Additionally, documentation is sparse and there are multiple ways to implement similar features. This being the case, much of our code was not directly translatable to Tkinter elements once time came to include that module. This meant some refactoring and a lot of piecing code together to make the system work as expected. Given the struggles we faced with the Tkinter package, it's clear that we would have planned things differently if we knew more about how to use it. We expected to wrap our back-end in graphical elements when in reality Tkinter needed to be much more intertwined with the main gameplay loop methods that were being executed behind layers of abstraction. This brings to light a challenge in trying to design and plan without getting into certain levels of specificity. If we had to scope out how we would use Tkinter earlier in the design process, things may have gone smoother. That being said, it seemed like we were intentionally trying to avoid that level of detail when reaching certain deliverables. This brings to light the critical nature of domain knowledge in planning steps. In the case that domain knowledge is absent, ample time for package/framework investigation should be set aside earlier in the process. If we were to do this again, we would get more specific earlier in the process so that there were less unknowns going into later stages of the software development life cycle.