Project Journal

Time Review

I would say that I worked about 100 to 120 hours total on this project. We met mostly every night and worked about two to four hours each night. Some of the nights, especially towards the end, we worked maybe eight to twelve hours each night. I started on this project the night of the day it was assigned and ended this project early in the morning of the day it was due but took off for basically the weekdays of Thanksgiving week, the week after, and the exam week after.

At the beginning our time was spent planning and writing the API. As features needed to be implemented, I spent my time pair programming with basically anyone who needed help on their part (first with Ben and the physics engine and hierarchy, then the authoring team, and then basically everyone). This involved planning for the design and approach for implementation, coding, testing little parts of the whole projects, bug fixes, and refactoring as needed. Towards the end of our project, we were still adding a few new features and trying to make our project look good as well as make bug fixes. The last two weekends of the project were full of all-nighters to make our project better.

I did not really personally manage my code seeing as how I was mostly pair programming. I would basically just code when we were all together (or a lot of us were together) and would basically then people that I would be pushing if it was going to affect someone's code (especially if I was in a branch that others were also working in). Additionally, on authoring, we would test by using the GUI whereas on the engine team, we might have tested by little a little test and seeing the visual effects.

The easiest part of this project was doing the initial designs for data and the physics hierarchy – yes they took a little time but once I understand the problems in and out, it was easy to figure out. The hardest part of the project was the last two weekends, where we worked tirelessly to make our project better. This was a very busy time for us all. Most of my time was used well but in one particular instance, we were hardcoding some games just to fully stress test our engine, which took a lot of time; however this did make me understand the actual underworkings of one a game would be made.

Teamwork

The team spent about 20 to 30 ten hours planning design, mostly in the beginning of the project. I feel that too much time was spent planning in the beginning – we should have started coding sooner; however, at the same time, I wish we did more planning as we were coding just to make sure that all sub-teams were on the same page, a problem that almost occurred a couple times.

Abhishek was one of the primary workers on the player module. He also worked on the Key Mapping Utility. Shreyas also was one of the primary workers on the player module. He is also involved on implementing conditions and actions within the engine module. Chris was one on the authoring module team. Will was on the engine module team. Davis was on the engine team and worked primarily on conditions and testing. Additionally, he created the SimpleDB Utility. Safkat was one of the primary workers on the data team, which also handled error

handling. Additionally, he worked on the physics engine and hierarchy, various parts on the authoring module, CSS styling on the player module, and the Key Mapping Utility. Arihant was on the engine module team and worked primarily on conditions, actions, and components. Arjun was on the authoring module team. Kevin was on the authoring module team. Eli was one of the primary workers on the data module team, which also did error handling. He also helped authoring with some features and created the GenericTypeAdapter GSon Utility. Benjamin was on the engine module team and worked on the physics engine, hierarchy, and actions. Wesley was on the authoring module team.

For the most part, communication was good among almost everyone on the team. During long nights, it was a little hard to communicate because of how tired we were but all of us being in the same building as we coded mitigated this fairly well.

The team's plan for this project was decided very early on and we kept it that way – the only thing we really did with the plan was to clarify it further and further as implemented more and more of the project. The plan was pretty stable – it did not change that much.

The plan held up well over the course of the project. Team roles were clearly defined at the beginning and for people on either the engine or authoring module, their roles stayed the same. The others (myself, Eli, Shreyas, and Abhishek) were on smaller modules that got done quicker, so we would move onto doing other things like helping the other two modules and working on utilities.

Commits

I personally pushed 16 commits worth 579 additions and 175 deletions on the master branch. The message of my commits are accurate of those commits but not of my work contributed to this project. One small reason for this is because some of my commits were to fix merge conflicts. Another much bigger reason is that I was pair programming with many of the others on the project. Additionally, I worked on testing, which didn't go on the master branch. I believe that a lot of my commits will be seen on other branches.

One of my commit messages reads "Dragging offscreen works for all sides." This was to show that a new feature was implemented. This feature was to make our initial demo for authoring look nice. It came at the right time, right before the demo. This commit did not, fortunately, cause a merge conflict for others but I did have to fix a merge conflict in order to push this.

Another one of my commit messages reads "Fixing ErrorPopUp to look nicer." This was to show that a new feature was added. This commit was done timely, a few days before the cod freeze. This commit did not, fortunately, cause a merge conflict for others because mostly I only worked on this.

One final commit message I will mention reads "Key mapping now fully working again. Just need to generalize to make it a utility." This is simply showing that a small bug was fixed and that now we just need to generalize it so other teams can use it as a utility. This was done somewhat not timely because this utility was added fairly close to the code freeze and thus other

teams weren't really given an opportunity to see and use it. This commit did not, fortunately, cause a merge conflict for others.

Conclusions

Even though I knew this project was large, I still feel like I under-estimated the size of this project. In the future, I can better estimate the sizes of projects by heeding the word of others that have done it before. I feel like I took on enough responsibility for the team since I was always there to help others in whatever way I could – I showed up to almost all meetings and every night we were going to code all night.

None of my code really needed editing since I was doing a lot of bug fixing myself. Pair programming also help me achieve this because we partners would catch each other's mistakes.

In order to be a better designer, I should keep reading the material assigned and deeply thinking about design before doing any coding. I do not believe there is anything I should stop or start doing.

In order to be a better teammate, I should keep communicating, thinking about design, and being able to helping out. I do not believe there is anything I should stop or start doing.

One thing I would do to improve my grade on this project is to implement some more features and bug fixes– both on the front- and the back-end. There are some features we did not fully implement due to a time constraint but we at least took an elementary pass at parts of those features.

Design Review

Status

The code in the entire project is not generally consistent in its layout, formatting, style, and descriptiveness. This is because there were a lot of people on this team working on a lot of different classes. Also, we did not take time at the end to format all classes because we cared more about getting features implemented. Additionally, a lot of the code is not readable, especially in the engine module and testing classes. Naming conventions are especially bad but documentation would help mitigate this.

There are a lot of natural dependencies in our code, mostly probably in the order of calls but I cannot get specifically into it because there is so much code and so little time. I believe overhearing that the collision handling in the game engine has some bugs because of the order of calls dependencies.

Features and new instructions are easy to extend or implement if they are simple, both in the engine and the authoring. Conditions and actions are easy to add as long as you write them but even that is quick and easy. So are physics components like new forces and the such. In the authoring module, they can easily mimic new features added in the game engine. This is very easy because they coded in a very modular way. Using the MVC design also helped this make it easier.

Most features could be tested for correctness on a preliminary pass by just looking at the GUI and running the program. It is very easy to do so. Testing game, however, is sort of hard because they are so complex. Making "cheats" mitigates the problem but it is still fairly hard.

I found a fair amount of bugs in our code because that is what I worked a lot on. However, one specific bug I found was how the ribbon on the player GUI was not working because fxml and css are a little finicky.

One class that I did not write but seemed interesting was the DataWrapper class. It is a purely passive data object. It is well written for a passive data object but it could use some documentation. It is interesting because this class is vital for serialization and derserialization. There are not many things I would tell the author other than giving some documentation. To extend it to other projects, one would have to simply change all of the classs to match the instance variables and getter methods that would be for that project.

Another class I did not write but wish to talk about is the Observable class. It is interesting because it is the same as the java Observable class except for one thing: the instance variables are now transient. This is done in order to be able to serialize and desesrialize some objects that we use but are observable. The code is very clear and well documented. Also it doesn't give other object control over it – it is the only class that can make changes to itself. Any other project could use this class if they chose to – it is basically the same as the java observable object.

One final class I would like to talk about but did not write is the GameObject class. This is interesting because it is a core of the game engine and our design – anything you want in a game is a game object. I do not believe it is that well written. If you wanted to hardcode a game, you have to create a new game object and then immediately set its identifier. This is very annoying – if you want all game objects to have an identifier, put it in the constructor! Also the class has a lot of getters and setters but I believe that was basically needed in order to change a game object. One good thing about the class is the clone constructor – it is easy to copy a game object in this way. I would tell the author to make the constructor take in everything it needs so I don't need to keep setting the identifier. Additionally, some documentation would be nice. Also, there is both an ID and an identifier, which is confusing as well. To put this in another project, the project would have to need use for a game object of some sort; otherwise, I do not see how other projects can use it.

Design

In our design, we have four modules: authoring, data, player, and engine. Authoring allows the creation and editing of games by a user. Its most important parts are the MVC design and extensibility through event handlers and observer/observable relationships. There is a big controller that talks between the many models and views. Extensibility comes through event handlers because anyone can add a new feature implemented in engine to authoring by just creating some handlers for it and a model and a view. Authoring calls upon the data module to save and load games into their environment. Data is simple: one class handles all reading and writing to file, no matter if it's a new/existing game or a save state. It also utilizes a utility that makes json serialization/deserialization easier by finding out what subclass implementation a superclass has. Player represents the front end GUI elements of the game. It calls upon

datamanger class to serialize (for progress states) or deserialize (for games) a datawrapper object, which holds all things necessary for game data. Player is further split into a model and a view. the view just shows the front end representations of the game. The model sets up the two-way communication between data and the one-way communication to the game engine, which provides the primary building blocks to create and run any game. It has gamemanagers, levelmanagers, physics, and rendering. It also has game object, which were explained above. The gamemanager initializes all the moving parts of a game while the level manager tracks and manages progression through the various levels of a game. Physics is a physics engine that uses a physics component hierarchy to simulate realistic physics. Rendering is what interacts with the canvas provided by the game player and generating the rendered nodes, or images, of the game objects to the screen. The game engine also uses condition and action pairs to create games, but this is further discussed below.

The game is basically a folder containing a json file, a folder with all the images, a folder with all the sound files, and a folder with all the progress save states. This is how your game is represented. To edit a game or create a new one, simply open it up in the authoring environment from the vooga salad bits please splash screen using main.java from the application package. To play the game, simply open it up in the player environment from the vooga salad bits please splash screen using main.java from the application package. To play the game,

My code is designed simply: the utility is a self-contained class pretty much other than for a css file. It builds itself and you can write inside of it whatever specifics you used to ties button/key events with whatever you want to do. The physics hierarchy is extensible to where you can easily add any sort of constant or vector or force. The datamanager class is simple as well: it takes in game data and a file path in order to write to a file; it takes a file path to read data from. The writing of progress save states is essentially the same as writing and reading authored games. Everything is contained within itself for datamanager.

One issue we dealt with in the very beginning was to even create a datamanager class or not. We could have just put it elsewhere but we decided against it. One other issue we faced was that of how to make our utility general. I wanted to make people code as less as possible but in order to do key mapping, the user has to fill in their implementation of conditions and actions. In order to do this, we basically leave one method in the class empty so a user can fill it in. we have to do this because we cannot know their implementation of conditions and actions. This is further described in the next section. There were really no issues with the other parts of the code I wrote.

One feature I did not implement was conditions and actions pairs, one of the cores of our game engine. The feature is what makes our game engine extensible: any condition can be paired with any number of actions. The user can also create whatever conditions and actions they want. All they have to do is extend Condition.java or Action.java.

One feature that I did implement is the limitation of dragging off the authoring environment for game objects. This code is in GameObjectDragHandler.java. If you put a game object on the screen, you cannot simply drag it off the screen, it will stay inside the viewable space of the level in the authoring environment. This was easily done using if statements. One bad part is that it always checks every side so there are four if statements but this needed to be done in order to check for multiple sides being crossed. If you wanted to use this code in a

different project, it would be easy – just replace the heights, widths, and objects with the corresponding ones you want to use.

Another feature I implemented is the keyboard mapping utility. The classes you need are KeyboardView.java, KeyMapForm.java, MappedKeys.java, keyboard.css, and form.css. Keyboardview.java is well designed because others can simply fill in one method and use this utility for their project. Additionally, all the methods are fairly small and each do one specific thing. Also, there are only two public methods – every other method is private. Also the keyboard is styled using css, which is extensible because anyone can make it look however they want. For KeyMapForm.java, this class is not as substantial but it is simple and well designed. There is one public method that creates the form. Additionally, it is extensible because the form is styled using css, so anyone can make it look like what they want. MappedKeys.java is not a great class however because we set up two arrays with the keys we want to use and their corresponding key codes. We wanted to do this in a resource file but that was messy because resource bundles don't save in an ordered list like we want.

Alternate Design

There really were no project extensions listed directly we wanted to implement new things after the basic implementation deadline; however, we knew what we wanted to implement so our design was already ready for those when we originally designed the program.

The original API did change but it only really grew. We added more classes and public methods as new features and abstractions were made. If we were to actually rewrite the API now, it would look differently but the basic underlying concept of the interactions would all be the same.

One design decision we made was to separate out the data module. This was in order to create a level of abstraction and to separate the engine and the authoring from talking to each other directly. The other option was to put the data somewhere in the engine or the authoring and then just have it talk to the other. I would go with our current implementation because if the authoring and the engine talked to each other, there would be some dependencies that would have been created. Additionally, the separation caused it to be cleaner and totally separates the two modules, which helps eliminate interference.

Another design decision we made was to do css/fxml styling and building in the player GUI. The other option was to just make the entire GUI in a java file. I believe that the current implementation is the best – yes, it creates dependencies on outside non-java files but it is truly extensible without having to writing any java code. This has helped us make our player GUI look good.

One final design decision I will mention is that we decided to create a physics hierarchy and then in the physics body to separate the methods to add vectors, forces, scalars, acceleration, and velocity. Originally, we wanted to have one method that would add any physics component but there was no good way to do this – we would have to use if statements and instanceof (very ugly!). Our current implementation is cleaner code yet more methods but it works well and looks decent.

One bug in our program is that collisions don't exactly work. The way to test it would be to play a game with collision conditions and see if the game is acting correctly. You could also put println statements in the condition and associated action to see if they are working correctly. I believe the problem lies with when the game goes into the player and is dependent on the aspect ratios of the display it is on. To fix this, there should be some relationship between aspect ratio and game size. Right now this is a hardcoded global variable – we must change this to fix it, I believe.

Another bug is that not all actions are implemented, reducing some of the functionality of the games we wanted to make. To test this, you must play a game with these actions. Additionally, to fix this, all one must do is to simply implement the actions.

One final bug that I will mention is that some of the data pathing is correct, especially for the change image action. The way to test this is to make a game with this action and see if it works – simply make a button press condition in one level of one game and you can easily test it. To fix this, all one must do is go into the code and see what the problem is – carefully examine and solve.

Code Masterpiece

My code masterpiece is the keyboard mapping utility. This includes KeyboardView.java and KeyMapForm.java. Keyboardview.java is well designed because others can simply fill in one method and use this utility for their project. Additionally, all the methods are fairly small and each do one specific thing. Also, there are only two public methods – every other method is private. Also the keyboard is styled using css, which is extensible because anyone can make it look however they want For KeyMapForm.java, this class is not as substantial but it is simple and well designed. There is one public method that creates the form. Additionally, it is extensible because the form is styled using css, so anyone can make it look like what they want.