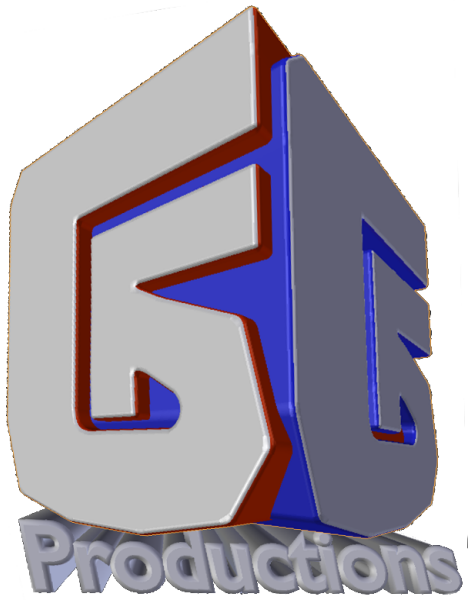
CS673F14 Software Engineering

Group Project - GG Productions

Project Proposal and Planning



Team Member	Role(s)	Signature	Date
Nabil Lamriben	Project Leader / Design Leader / Environment & integration Leader	Nabil Lamriben	9/16/14
David Lustig	Backup Project Leader / Implementation Leader	*David Lustig*	9/18/14
James Raygor	QA Leader	*James Raygor*	9/17/14
Jeannie Trinh	Requirement Leader	*Jeannie Trinh*	9/16/14
Paul Pollack	Design Leader/ Configuration Leader	*Paul Pollack*	9/16/14

Revision history

Version	**Author**	**Date**	**Change**
1.0	GG Productions	9/25/14	Initial draft

Table of Contents

1. Overview

It is well known that if a child does not find learning software fun to use they simply will not use it. In this sense, software that advertises itself as educational yet is panned by its target audience is somewhat paradoxical, for you cannot teach someone if they are not paying attention. Thus it can be concluded that, for educational software to be successful, it must engage children over the long-term while simultaneously teaching them. Our team has set out to develop a product that meets these requirements: Letter Storm, a game that is both entertaining and educational.

Our game allows a player to control a character who can move and shoot projectiles in the style of a traditional 2D vertical-scrolling shooter. We have incorporated elements into our design from this genre such as enemies, obstacles, power ups, bosses, and a running score. This is not done at the expense of educating the user, though, for learning challenges are also tightly incorporated. The user must collect letters from defeated enemies and use them to defeat bosses by spelling words correctly when prompted.

Our intent is to make the educational component of the game extremely customizable, so that users may input words geared towards the comprehension level of the player. While Letter Storm can be used by anyone, the target audience is early elementary school students. In spite of this, it will likely be the case that the heavy incorporation of shooter mechanics will appeal to children of all ages.

2. Related Work

DoDonPachi
DoDoPachi is a series of vertical-scrolling 2D shooters developed for arcades. [1] The games are geared towards serious fans of the genre and often praised for their challenging gameplay. [2] In fact, this subgenre of shooters is often called a “Bullet Hell” because you’ll find more pixels on the screen occupied by enemy bullets than not.

Our interest in this game is not rooted in its difficulty, but rather as a case study of excellent game design. For a series that has received acclaim for over a decade as a landmark in the genre, it serves as an ideal model to follow when considering design tasks such as enemy spawning, movement and firing patterns. Letter Storm is not intended to be anywhere near as difficult as the DoDonPachi series since our target audiences are radically different, but the patterns alluded to above are universal to shooters and there is no need for us to reinvent the wheel.

Super Stardust HD
Super Stardust HD has often been referred to as Asteroids reimagined for the PS3. In this 2D, top-down shooter, the player pilots a spaceship over the spherical plane of a planet, defending it from incoming asteroids and alien spaceships. [6] Its combination of a simple control scheme, straightforward goals, weapon-based strategy, and clever use of 3D space make it a 2D shooter that has earned the praise of many modern gamers.

Of all the related works considered, this is the most modern. It is clear that its designers were thinking outside the box when they developed the game’s aesthetic, putting a fresh spin on a well-worn gaming genre. Likewise, entertainment value was also enhanced by their strategic use of weapon types and powerups. We hope to incorporate some of these design concepts into Letter Storm to make it not just a learning game, but a game that is fun and memorable to play.

Z-Type
We are not the first to incorporate typing elements into the shooter genre and we probably won’t be the last. Z-Type serves as a particularly interesting comparison with our own project simply because it illustrates how much potential there is for fresh gameplay even with the same basic concept of a “typing shooter.” In Z-Type, enemies fall towards the player, who is stationary on the bottom of the screen. Each enemy has a word associated with it that the player must spell to defeat. [3]

Letter Storm differs from Z-Type in several ways. Gameplay is divided into two basic stages. The first stage consists of defeating enemies and collecting letters, and the second consists of using those letters as ammunition to correctly spell one word associated with a boss. Where Z-Type requires the player to type quickly and without much discouragement from making mistakes, Letter Storm requires the player to know how to spell words with minimal hints. The focus is not speed, but rather accuracy. Z-Type is targeted towards an audience who already knows how to spell and for the most part, knows how to type correctly as well. Letter Storm makes no assumptions about a

player's abilities as a typist, and ideally has players spelling words that they didn't previously know how to.

Pocky & Rocky

We believe it is important to incorporate a unique and fresh aesthetic to Letter Storm that helps set it apart from similar games as well as cater to our target audience. The majority of 2D shooters adhere to a number of cliches, in that they typically feature the player piloting a fighter jet or spacecraft blasting away enemy tanks, planes, and alien motherships. Besides the fact that this design approach is significantly overused, we don't believe imagery that conjures themes of warfare, destruction, and violence are befitting of an educational game.

Pocky & Rocky was a hit shooter for the Super Nintendo that featured a very bright, cutesy aesthetic with colorful sprites and forest backgrounds. [4] Players may control either a shrine maiden or a raccoon, two character archetypes not often seen in shooters. We hope to take a page out of Pocky & Rocky's book, and although its gameplay strays further from a traditional shooter than we intend to, the lighthearted aesthetic is a valuable source of inspiration.

3. Detailed Description

This section describes the minimal requirements of the product. Requirements are listed in bullet-form to maximize readability and organization. Additional details can be found in the project's Pivotal Tracker project, found at https://www.pivotaltracker.com/s/projects/1170210.

- General
  - The product must be a game that is playable and appropriate for children ages 5 and up
  - The game scenarios presented must enable the user to spell words (learning component)
- Target Environment
  - Product must be deployed as a standalone desktop application (Windows).
  - Optional: the product should be runnable on the web (html) (currently web version is runnable but not supported; features are not guaranteed to behave correctly)
- Customizability
  - Game must provide an option to allow the user to add a custom word list or dictionary, which can be used as part of the game
  - Game must provide one or more default lessons that can be used as part of the game
  - Game must provide one or more characters for the user to choose from for gameplay
- Interface
  - A keyboard must be the primary method of input for using the software.
  - Optional: mouse or mouse-keyboard combination as input for the software.
  - A menu system must be provided for general navigation of the game
    - Main Menu
      - Play Game
        - Select Enemy Difficulty
        - Select Word Set/Lesson
      - Tutorial
      - Story
      - Manage Lessons
      - Quit Game
    - Pause Menu
      - Return to Main Menu
      - How to Play
      - Quit Game
      - Settings for Music On/Off
  - A HUD must be displayed in-game to assist the user in keeping track of information required to play the game, such as health and letters collected
- Functionality
  - Game must be able to store the user-defined word sets/lessons in a save file (implemented but not tested)
  - Game must be able to load user-defined word sets/lessons from a save file (implemented but not tested)

- Design and Strategy
    - Game must provide enemies for the player to avoid or defeat
        - Standard enemies
            - Passive enemies must drop consonants upon defeat
            - Active enemies must drop vowels upon defeat
        - Bosses must correspond to words that should be spelled
    - Game must provide collectible game items that will aid the user in beating the game
        - Power-Ups
        - Letters
    - Game must allow user to defeat standard enemies using one or more weapons
    - Game must allow user to defeat bosses using the letters acquired from all enemies defeated earlier in the level

4. Management Plan

a. Process Model

The software process model to be used in the development of our product will be primarily based off the Spiral Model. The overall process flow will proceed sequentially through four (actually, five) steps every iteration of the development cycle, as detailed below. Due to the short cycle time, slight variations of scrum communication practices will be used to ensure tight collaboration between all team members.

Overall process flow (per cycle)

- Step 1) Requirements Gathering: For first round, brainstorm user stories and document them in Pivotal Tracker. For subsequent rounds, demo prototype to customer to obtain clearer requirements.
- Step 2a) Risk Analysis: Order requirements based on priority and feasibility, and tackle those with the highest first
- Step 2b) Prototyping: Develop or enhance a prototype of the product based on the most feasible, highest priority requirements not yet met
- Step 3) Testing and Evaluation: test the prototype for bugs and defects
- Step 4) Plan next phase: Update documentation and requirements' ranking based on test results

Communication

- Team members will send daily scrum emails to the team to ensure fluent team communication
- Weekly meetings to determine goals for the week, with occasional biweekly meetings to ensure we are on target to meeting our goals

b. Objectives and Priorities

As with any software engineering project, the highest priority is satisfying the customer. To do this, at least the most important requirements should be met within the allotted time. This can only be done if all people working on this project invest enough time into it to meet the goals of each cycle. This is more likely to occur if team communication is high and subject matter experts are willing to assist other members with issues relating to their area of expertise.

Thus, the objectives and priorities for our team are to:

i. Release a new, customer-testable version of the product at the end of each iteration (every three weeks) that incorporates the requirements scoped for that iteration
ii. Develop a final product that meets at least the base user requirements, if not better
iii. Ensure fluent team communication using the SCRUM process

iv. Avoid investing more resources in meeting a requirement than the requirement is worth
v. Create an environment of tight team collaboration, such that the skills of one member who is an expert in a subject can be utilized to prevent other members from wasting time on troubleshooting that subject

c. Risk Management

Requirements will be evaluated against their feasibility and complexity to determine the project's initial risks. During the process of developing and testing the product, additional risks will become apparent. All risks will be documented within a risk categorization table. The severity, impact, and potential solutions to unresolved risks will be re-evaluated by the team at the start and end of each iteration.

Risk categorization table
A risk categorization table will be maintained for the duration of the project.

Example:

risk	rating	source	Impact	solution
Blender export to Unity problems	3/7	integration	Deadline push back	Do test prototype, then generate a template for exporting from Blender to Unity
Letters visually vanish, but the letter objects are not deleted properly and are causing memory leaks	1/7	Implementation	Features branch is abandoned to focus on a fix	Review code and do memory testing before new iteration starts

1- Risk: Identify the risk that we may encounter
Example:
Missing team member risk - Team member is not able to deliver a task due to higher obligations.

2- Rating: Will use the same scale presented in *Quality assurance plan* , *defect management* section e.

3- Source: The source categories reflect our team members role: Design; Integration configuration; implementation (or coding) ; QA

4- Impact: The event's impact on the overall lifetime and quality of our product.

5- Solutions:

- Solutions can be found by organizing a brainstorming session with teammates, or learned from post-performance analysis.
- Post-performance analysis: At the end of each week, the team will meet to discuss the problems that were encountered and brainstorm solutions. This will decrease the chance that the same mistakes are repeated in the future.

d. Monitoring and Controlling Mechanism

1) Monitoring and tracking

Project tracking consists of comparing the project plan with the actual advance of the project.

- Determine Work Done: the % of task completed.
- Determine Resource spent: the time and other resources spent on a specific project
- Compare Work Done vs Resource spent
- Track Milestone: represents important achievement in a project. We will be counting the completion of Prototype 1 as milestone

2) Control Mechanism

- Project Meeting:

Vital to a healthy team development. The team will meet at least once a week face to face.

- Individual catch ups with team member:

More granular issues can be solved in a one on one meeting. Either via skype, or face to face. This type of meeting will be very important for sub groups such as the two person code review team, and the 2-3 person Visual design team

- Collaborative contribution to issue and risk register:

We will be using Pivotal tracker as a monitoring mechanism and risk register. Team members will update user stories and issues encountered on an individual basis.

- Timely completion of timesheets and daily scrum email:

Team members will update the weekly report, and the daily scrum email.
Scrum email usually consists of a summary of work done the day before, a summary of the work planned to be done that day , and any blocks encountered.

- Percent done:

At the end of each iteration, we will compile a task % done

e. Schedule and deadlines

The product development process will take place over 4 cycles, or iterations. Each cycle will span the length of 3 weeks. In general, steps 1 and 2a of the spiral model will occur during the beginning of week 1, steps 2a and 3 will occur in tandem from week 1 to 3, and step 4 occur during the end of week 3. The primary exception to this is the first cycle, during

which most time will be spent gathering requirements, setting up the development environments, and writing the first draft of all primary documentation.

The known deadlines and dates of interest are as follows:

- 9/04 - First iteration begins
- 9/25 - First iteration ends. SPPP and Product Presentation are delivered to customer. Second iteration begins following customer feedback.
- 10/15 - Second iteration ends. Initial prototype of product is delivered to customer. Third iteration begins upon the reception of customer feedback.
- 11/06 - Third iteration ends. Beta version of product is delivered to customer. Fourth iteration begins upon the reception of customer feedback.
- 12/04 - Fourth iteration ends. Final product and corresponding documentation is delivered to customer.

5. Quality Assurance Plan

a. Metrics

The primary metrics of interest will be recorded within a Metrics Tracking spreadsheet administered by the QA leader. Metrics applicable to user requirements may also be logged in Pivotal Tracker. Metrics will be reviewed regularly to obtain a better understanding of how the product and the development proceed.

The metrics of interest include:

- Product-related metrics
  - Size: number of lines of code
  - Quality Level: number of defects per 1 thousand lines of code
  - Customer Problems: number of issues customer has with program
  - Customer Satisfaction: based off weekly survey, 1-Not at all satisfied, 2-Slightly satisfied, 3-Moderately satisfied, 4-Very satisfied, 5-Completely satisfied
- Process-related metrics
  - Customer Satisfaction: based off weekly survey, 1-Not at all satisfied, 2-Slightly satisfied, 3-Moderately satisfied, 4-Very satisfied, 5-Completely satisfied
  - Defect Repair Rate:number of defects repaired per a week

b. Standard

Reference “GGProductions Code Standards using C#” however here is an example

> “do use **camelCasing** for method arguments and local variables.

```
1. public class UserLog
2. {
3.     public void Add(LogEvent logEvent)
4.     {
5.         int itemCount = logEvent.Items.Count;
6.         // ...
7.     }
8. }
```

”

Why: consistent with the Microsoft's .NET Framework and easy to read.

c. Inspection/Review Process & Table

Any and all C# code is subject to review. Code reviews should be executed within two man code teams. Where the coder exams the code prior to handing it off to their coding buddy, who will review and verify the new code. This type of review should be conducted after every upgrade, push or commit to github in order to ensure defects are kept at a minimum. Time will be set aside for Weekly team meeting reviews and inspections. If defects are found the coding buddy/QA lead will fill out the Inspection doc, how it was found and update the information to the Defect Management Table or to the coder.

d. Testing

Types of Testing:

- ❖ Unit testing: testing each function or block of code by verifying the operations work (example: are if/else statements working correctly)
- ❖ Endurance testing: what happens when the game runs for a 15 min, 30 min, 45 min and 1 hour
- ❖ Compatibility testing: testing to see if the program runs on the proper browser or desktop format
- ❖ Game testing: aspects of the end user experience
- ❖ GUI testing: hows is the GUI and does it work as planned

During group and two person code team testing the following steps will be executed for inspection and review:

1. Planning: which type of testing will be executed
2. Preparation
3. Inspection Meeting: participants discuss what happened and how to adjust
4. Rework: execute solutions
5. Follow-up: did the rework solve the issues
6. Improve Process: can the solution be improved upon

The Inspection doc will be as such:

- ❏ Who: QA leader and Team
- ❏ When: Date and Time
- ❏ What type of testing:
- ❏ What was found: (both positive and negative)

- ❏ What was changed:
- ❏ Pass/Fail:
  If fail go into Defect Management Table and proceed to fill out the Defect classification of Priority, Severity and Type

e. Defect Management Table

First Classify Defect using 3 levels

- ❖ Priority:What order will defect be taken care of
  High: Taken care of first
  Medium: second
  Low: third

- ❖ Severity:How serious
  High:Game Crash
  Medium:Game slowed
  Low:Game keeps running

- ❖ Type: What kind of problem or feeling you have about the issue
  High: Something is missing
  Medium: Almost working
  Low: Can move on without fix

Fill out the following information in the Defect Management Table

- ❖ Reference the Inspection sheet for the particular bug
- ❖ Give any information not logged in the Inspection sheet
- ❖ Date/Time
- ❖ Who found the defect

Repair/ fix section of the Defect Management Table

- ❖ Who is working on the fix for the defect
- ❖ Who is checking and verifying the fix has succeeded in repairing the defect
- ❖ Did the fix work: Yes/No: Yes move to the next defect, No the defect is then reprocessed into the Defect Management Table

This data will then be logged with Github, Pivotal Tracker and a Defect Management Table. Which will be administered by the QA Leader. When coders need work they can access any of the above tracking platforms and work on any of the defects they feel best suited to repair.

Order of Priority is as followed

Order of priority	PRIORITY	SEVERITY	TYPE
1	High	High	High
2	High	High	Medium
3	High	Medium	Medium
4	Medium	Medium	Medium
5	Medium	Medium	Low
6	Medium	Low	Low
7	Low	Low	Low

6. Configuration Management Plan

a. Configuration items and tools

Tools

Unity (Version 4.5) - Game Engine and IDE
Visual Studio 2013 (Express or Pro) - IDE
Git (version 2.1.1) - Distributed Version Control System.

Configuration Item Management

The name of our project's root directory will be **LetterStorm**. All source code, documentation and project assets including materials, models, prefabs, scenes, sounds and textures will be configuration items (CIs) tracked by Git. User and project settings files generated by Visual Studio and Unity will not be considered CIs, and specified in the .gitignore file so that they will not be tracked.

All members maintain ownership over a shared "central" remote repository named `origin`. The term central is used here only conceptually since Git is a distributed system. All team members will fork this repository and follow the convention that their forked remote shall be named after their own first name. In the case that two team members are working with a CI as a subteam, they may fetch from each other's repositories without modifying the baseline maintained in origin until their code builds properly.

b. Change management and branch management

We will be using a Git branching model based on Vincent Driessen's as described on his personal blog. [1] The model defines several branches and strictly enforces a well-defined purpose for each. The types of branches are as follows:

- master
- develop
- feature
- release
- hotfix

The Main Branches

Our two main branches are `master` and `develop`. As mentioned earlier, `origin` serves as our central "truth" repo. Everyone pushes and pulls to `origin`. On the `origin/master` branch, the source code of HEAD always reflects a production-ready state. That is to say, `origin/master` points to the latest release of Letter Storm. For the purposes of this project, it is safe to say that we will have exactly one commit to `master`

that corresponds to each iteration of the project. On the `origin/develop` branch, the source code of HEAD always reflects the current development baseline. All commits to this branch should reflect code that builds but is not yet considered stable enough for a production release.

The main branches discussed above live indefinitely. As development progresses, it would be extremely difficult to keep track of changes and support parallel programming with just these two branches though. To solve this issue, we will make use of three types of supporting branches: feature, release and hotfix branches.

The Supporting Branches

Feature branches allow us to start work on new features that may or may not be included in the upcoming release. A feature may be included two versions down the road, or perhaps it will be abandoned altogether and discarded. In any case, creating a separate branch to work on features ensures that our `develop` baseline is always in a working state.

Feature branches always branch off from `develop` and merge back into `develop`. Their names should reflect the functionality of the code. For example, if a team member began writing the code for an achievement system, an appropriate name for his branch would be `achievements`.

Release branches are created when code in `develop` is stable, ready to be merged back into `master,` and tagged with a release number. Instead of merging code from `develop` directly, however, we create a new branch. This allows one team member to put the finishing touches on the release while the rest of the team can continue working on `develop`.

Tasks that may be performed on a release branch include preparing metadata and fixing minor bugs. All features targeted for the upcoming release must be included when the release branch is created, otherwise their inclusion will be postponed until a future release. In general, release branches are to be merged back to `master`, but in the event that any minor bugfixes have been committed, they may be merged back into `develop` as well. The naming scheme for a release branch is `release-*`, where * should be replaced with the corresponding version number of the release.

The last type of supporting branch we will be using is a hotfix branch. These branch off from `master` and merge back to both `master` and `develop`. Hotfix branches are created to deal with bugs that have been discovered in an active production release and must be fixed as soon as possible. The naming convention for hotfix branches is `hotfix-*`, where * should be replaced with the corresponding version number of the release.

An important final note: If a release branch exists when a hotfix branch is created, the hotfix should be merged back into `master` and then into `release`, not develop.

c. Code commit guidelines

When writing commit descriptions, use present tense.

Committing to `master` is equivalent to releasing a new production version. The only time commits should be made to this branch are if they are merged in from either a release or hotfix branch. All such commits must be tagged.

When merging a supporting branch with `develop` or `master`, always use -no-ff in order to ensure a new commit object is created, even if the merge could be performed by a fast-forward. For instance, if a team member began work on an achievement system feature, he may create his branch as such:

```
$ git checkout -b achievements develop
```

This creates a branch named `achievements` that branches off from `develop` and immediately points HEAD to this branch. When the team member has finished writing the code for the achievement system, he can do the following to merge the feature into `develop`:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff achievements
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d achievements
Deleted branch achievements (was 05e9557).
$ git push origin develop
```

This way, we retain important historical information about the commits from `achievements` even after the branch has been deleted. We can safely revert back to an older version should we need to.

The version of the initial commit to `origin` will be 0.1. Subsequent releases will be tagged as 0.2, 0.3 and so on. Hotfixes will be tagged 0.1.1, 0.1.2 and so on. Incrementing the first number is reserved for the addition of an adequately significant feature to be determined in the future.

7. References

[1] Cave. *DoDonPachi*. Atlus, 1997. Arcade, Sega Saturn.
[2] Spencer Johnson. *DoDonPachi Introduction*. Hardcore Gaming 101, April 30, 2014. http://www.hardcoregaming101.net/dodonpachi/dodonpachi.htm
[3] Phobos Labs. *Z-Type*. Phobos Labs, 2011. Web.
[4] Natsume. *Pocky & Rocky*. Natsume, 1993. Super Nintendo Entertainment System.
[5] Vincent Driessen. A Successful Git Branching Model. January 5, 2010. http://nvie.com/posts/a-successful-git-branching-model/
[6] Housemarque. *Super Stardust HD*. Sony Entertainment, 2007. PlayStation 3.

8. Glossary

Develop (branch)
Branch reserved for ongoing development with infinite lifetime.

Feature Branch
A supporting branch reserved for the development of a new feature to be merged back into `develop`.

GG Productions
The alias for the team developing Letter Storm.

Git
An open source, freely available source-control system.

Hotfix Branch
A supporting branch reserved for fixing bugs as they arise in production releases as soon as possible.

Letter Storm
The learning game being developed with the goal of being both entertaining and educational.

Master (branch)
Branch reserved for production releases with infinite lifetime.

Pivotal Tracker
A web site used for documenting, ranking, and assigning user stories (requirements). The Pivotal Tracker project corresponding to this software development effort can be found at https://www.pivotaltracker.com/s/projects/1170210

Release Branch
A supporting branch reserved for putting finishing touches on a new release before merging with `master`.

Unity
A .NET-based game development IDE. http://unity3d.com/

Visual Studio
Microsoft's IDE for developing all things .NET. We will be using this to write the scripts backing our game.

Blender 2.7
Blender is a free modeling software. We will be using it to generate 3d game assets as well as creating animation sequences for those assets

Gimp 2.0
Gimp is a free image processing software very similar to Adobe Photoshop. We will be using Gimp to create all the 2 dimensional graphics such as textures, splashscreen, and hud components