Zan Balcom: zbalcom
Autumn Blackburn: kblack37
Jake Chiang: jchiang2
Alex Davis: ahd2112

Verigames: Crowdsourcing Code Verification Through Video Games

Motivation

Computer programs have defects, which, as history and experience shows, can be very costly when they survive to be in the final product. For example, on June 4, 1996 the Ariane-5 rocket experienced a launch catastrophe due to an error caused by casting 64-bit floating point values to 16-bit signed integers[3]. Program verification is the process by which a program is proved to be free of some of these costly bugs; however, it is expensive to have trained programmers spend the time to evaluate every line of code looking for these defects. This is the way nearly all verification is handled today. There are existing tools that can perform certain amounts of verification checking, but verifying a complete and meaningful program with these tools would take years even with a team of dedicated engineers. Due to these constraints, usually only the most important components of large codebases are verified.

Verification Games serve to transform the intricate task of code verification into an accessible game that can be completed with no special training, where finding a solution to a level provides valuable verification insight[1]. This game can then be deployed to a large audience, crowdsourcing the laborious task of code verification to a population that doesn't need any specialized education. Flow Jam (fig. 1), is one such game developed by a team at the University of Washington that uses a series of differently-sized pipes and colored liquid to represent the variables and values, respectively, of a program. Game levels are made by reading a Java program and a specification in the form of a type system and then converting these into corresponding system of pipes. The goal of the game is to manipulate the width of the pipes to produce a configuration that allows the fluid to flow through the whole system. Such a solution

Figure 1: main menu of Haxe port of FlowJam

equivalently is a proof that the program holds to the specification. One example of this type system in practice would be to look for null pointers in the program, with one width of pipe representing nullable assignments and another being non-null types. By solving the game, the player would be proving that no null value could be assigned to something that should not be null. If the player cannot find a solution, then they can use a "buzzsaw" to make all the fluid flow through the pipes. This in turn signifies that there could be a type error in the code that the pipes represent; a trained software engineer can then look at that part of the code and resolve any errors they find manually. In other words, the buzzsaw helps identify areas where a configuration of the type system could not be found that solves the puzzle, which indicates that the specification is not yet met.

With a crowdsourcing approach to code verification, the task of identifying potential problem areas in the code for trained engineers to review becomes much more manageable. A current issue with code verification is that it is a laborious project that has to be undertaken by trained engineers and thus verifying 100% of a large program correct would take far too many people-hours of work to consider as an option. With verification games, large swathes of the program can be verified correct automatically by players and so the developers of the program can then utilize their engineers more efficiently. Currently, however, these verification games are only capable of representing small programs because the number of variables, values, and dependencies quickly becomes unwieldy in meaningful programs. As with Pipe Jam this limits the power of the tool considerably since the number of pipes in larger programs would quickly become unmanageable for a player to deal with.

Related Work

Other attempts have been made to create code verification games with different approaches and results. One is Paradox[18][15], which operates on the same principles as Flow Jam, although Paradox handles the complexity of large programs by leaving it up to the users to decide how much of the program they want to verify. Fig. 2 shows a game of Paradox being

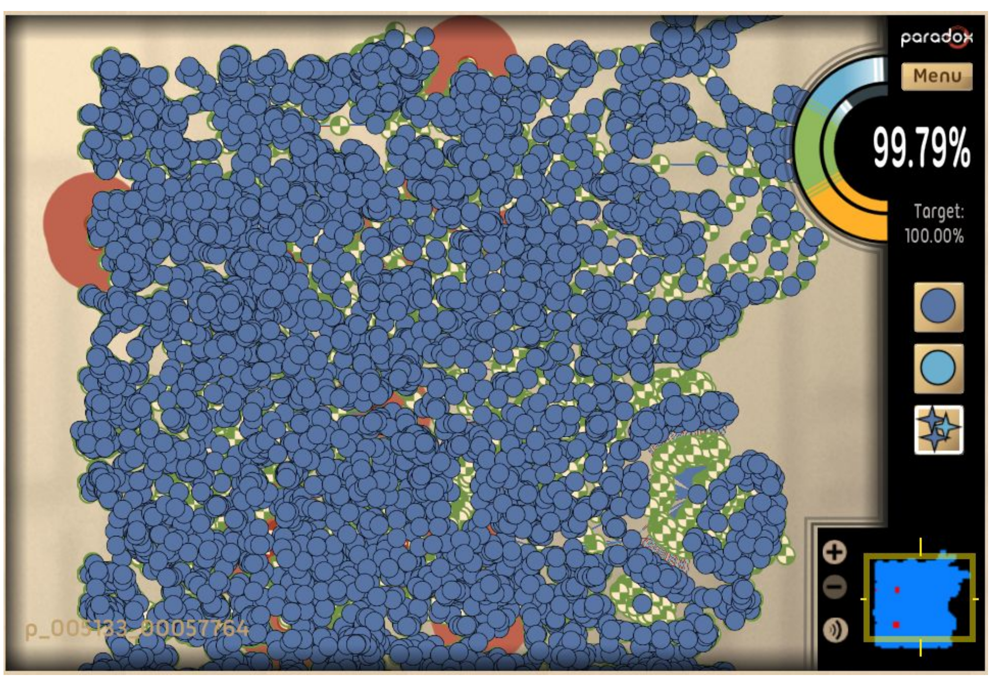


Figure 2: Paradox full view

played. The red areas identify constraints that are not currently met by the program and it is the player's job to change the links between the small circles to meet that constraint. Although this solves the same problem as Flow Jam, it doesn't abstract the complexity of a program in a meaningful way. While in Paradox you can use the map in the bottom right hand corner to zoom in on an area to work on, in Flow Jam the user is only given a small area to work on at a time that is connected with other areas out of sight. We believe Flow Jam's way of obfuscating the complexity of the program is important to keeping players motivated to play the game and in drawing in new players for the crowdsourcing of verification.

Pipe Jam is another project that is built on the same system as Flow Jam: it parses programs and constraints into JSON files that the game engine uses to construct its levels. This game represents variable assignments with pipes that lead from one screen to another to represent where variables are referenced and checked to see if they meet the constraint. The balls that run through the pipes represent those variables. If a ball is too large to fit into a pipe that represents the constraint, then that part of the program does not meet the specification. This is essentially the same mechanics as Flow Jam with a different skin, but we believe Flow Jam to be more aesthetically appealing than Pipe Jam.

Binary Fission[17] was created by researchers at UC Santa Cruz to move the task of identifying cyber vulnerabilities into the realm of crowdsourcing. From what little we have been able to find on it it seems to work by having players identify loop invariants that are hard for computers to recognize. Players of the game use different filters to sort out "quarks" in the game and from this they are filtering which invariants are true about loops in the code.

Besides code verification, games have been used to solve other kinds of difficult problems with success. A great example of this is Foldit[12], a game which has human players competing against each other to see who can fold a protein into the most stable shape. Prediction protein folding or designing new proteins that have a certain structure is an NP-hard problem and thus not practically solvable by computation alone. Another is Mozak[13], a game with the goal of deepening the understanding of neuron structure.

An effort to create verification games spearheaded by DARPA was released in 2013[[18][10], but the links to the games mentioned on the website in the press release now point to broken versions or bad gateways. Currently Paradox is the only working verification game we could find. The research behind the games that were developed with the DARPA project points to the main flaw that was present in all the games. The problem is that the games of this type may not be very fun to play. In any other game there is a development team that is designing the experience they hope the player will have, but with automatically generated verification games there can be little to no control over how the game is represented beyond that which is done automatically.

There also exist other tools for code verification besides games. For example, there is the Checker Framework[16], which Flow Jam uses as a backend to create the JSON level files. The Checker Framework's verifier is sound for most type annotations, like verification games, but also tends to create more false positives in its attempt to be entirely correct. Verification games, in contrast, utilizes people's innate problem solving abilities to reduce the amount of false positives it finds.

Approach

Our approach to verification games is to focus on distributing the game to as large of an audience as possible. This will place Flow Jam into the realm of large scale crowdsourcing: previous applications of crowdsourcing in computing have utilized many different amounts of users, from relatively small number of individuals like TopCoder to 75,000 users in the protein folding solver FoldIt[2]. To this end, we have developed a few different ideas to implement in sequence, starting with updating the current codebase to a new platform to improvements in the game to improve player retention.

The first goal is to update the platform that Flow Jam was built on. The original game was written in ActionScript3, which has fallen largely out of favor in the game development world as Adobe has announced that it will stop supporting its Flash Player browser plug-in[4]. We plan on updating the codebase to Haxe using the as3hx library so it can be easily configured to be distributed via iOS, Android, or HTML5, reaching a wider audience[8][9]. In addition to this, many of the external libraries the original codebase depended on have fallen out of use or into disrepair, so we will have to find or create replacements for those.

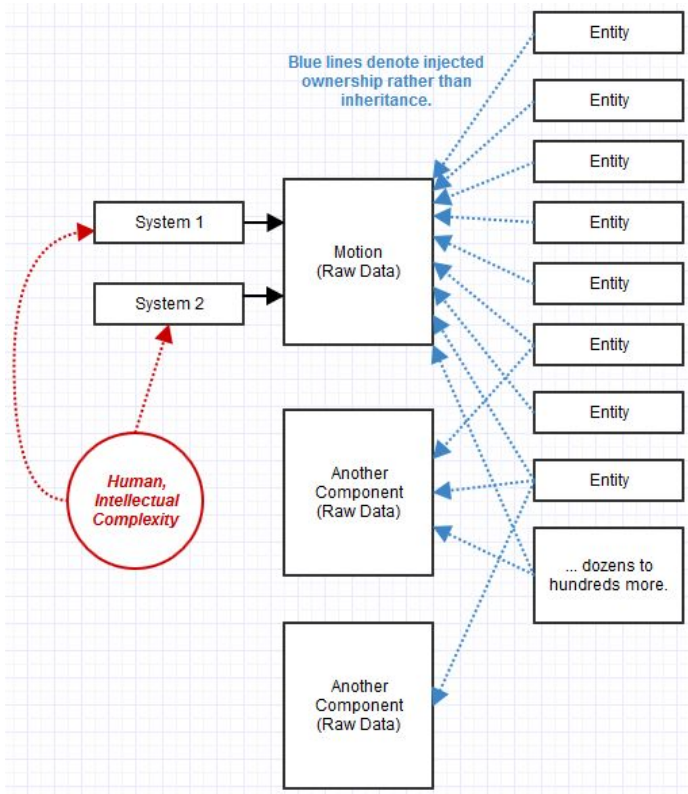
The second goal is to, after the codebase is fixed and we have the prototype of the game up and running, begin playtesting Flow Jam. We will have a decent amount of people (at least 20) - including those both within and without of our social circle (recruiting from, for example, the HUB on the UW campus) - playtest the game and observe their reactions and ask them about their experience with the game. From this, we can pick components of the game that has the greatest potential for player engagement and thus will most likely reach the largest audience(more information on this testing can be found in the Playtesting part of this document).

The last goal is to, once we know more about the playability of the game, implement new features to the game to increase player engagement. Secondary objectives have been shown to increase player engagement when placed on the path to completing a level (as opposed to out of the way of the intended path)[5]. To this end, we plan on implementing a system of secondary objectives in the form of collectible tokens that are obtained in the normal path of solving a level. This will be integrated into the scoring system, with picking up collectibles being another way for players to increase their score. With this scoring system in place, we can also implement a competitive aspect into the game, such as a score- or speed-tracking system so players can compete with their friends and possibly recruit new friends into playing[6]. Many of the most widely-played mobile games today have a competitive dynamic[1] (whether directly against another player or in a form similar to high-score charts) so we are confident that this is a worthwhile venture. We want to retain the simple-to-grasp nature of Flow Jam and not make the game too unapproachable for newer players, so we are limiting the number of planned features to avoid feature bloat.

---

[1] Ascertained by a familiarity with and an examination of the top-played games of all categories on the Apple App Store.

Architecture



[7]

Figure 3: typical entity-component-system architecture

We wish to move the codebase away from its current, less flexible and maintainable architecture to an entity-component-system architecture (fig. 3). From its current state, the code will require a lot of refactoring. Since it was designed as a prototype and not a codebase to be maintained long-term, it has no definitive architecture that would allow for interesting feature growth. Many classes (Level, World, GridViewPanel) are so-called “god classes”, performing far too many tasks and having far too large of a stake. As well, many classes make extensive use of static variables to share information. There’s also no clear recognizable architecture. For these reasons, we believe a refactoring is in order, though new features are our first priority.

The goals of migrating the codebase to an entity-component-system architecture are ensuring new game systems can be added more easily as well as reducing the current complexity of the code (for example, the “god class” mentioned above, scenes.game.display.Level, is over 2000 lines of code and performs many different tasks that could be refactored into other classes). ECS architectures allow for easier development of new game systems because adding one in is (almost) as simple as creating a new type of component and a system to govern it. They also reduce the complexity of the code through the principle of defining objects through composition instead of inheritance.

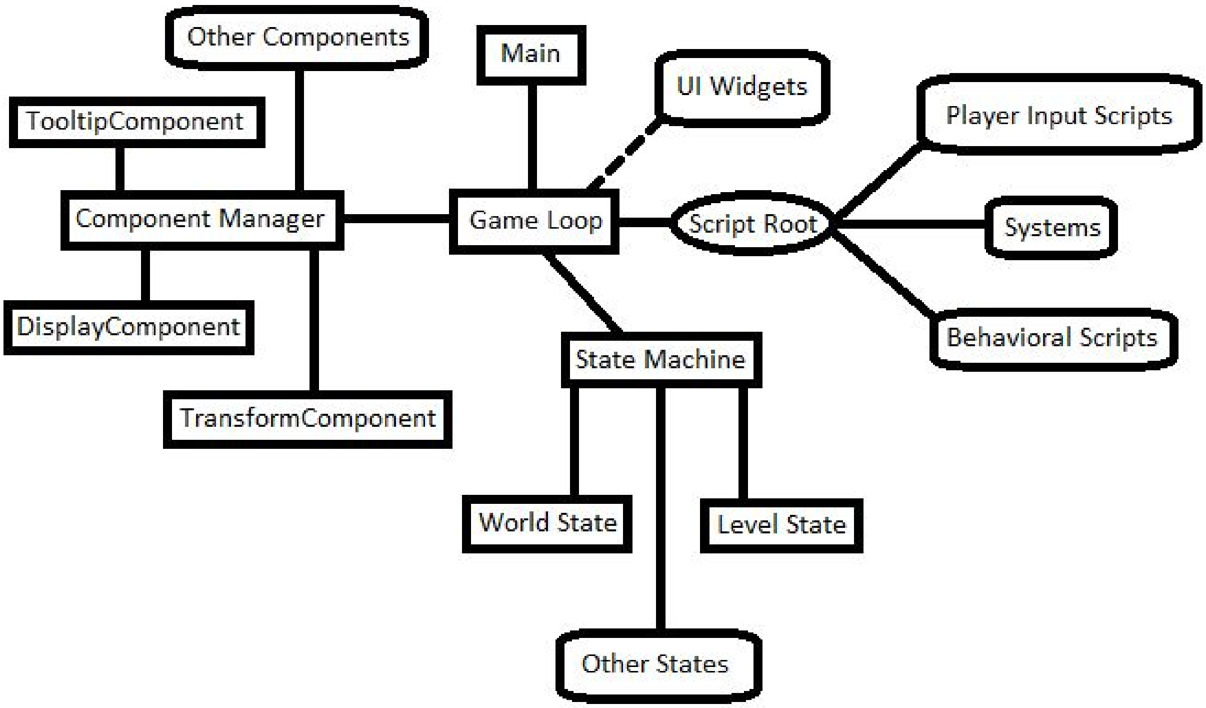


Figure 4: new high-level architecture

Above (fig 4) is the planned architecture at a high level. Main refers to the static entry point, where possibly different versions of the game (for different release platforms) can be instantiated.

The game loop is not quite a class itself, but part of a class instantiated by Main. It contains many different elements of the architecture and serves as the central locus of communication between them. It passes itself down to classes that may need to reference resources, such as scripts that operate on some particular entities.

The component manager is one element of the game loop class. It handles all addition & removal of components from entities as well as providing methods to retrieve components of a certain type or that belong to a specific entity. Components are generalized objects that systems can operate on; for example, a transform component contains an entities transformation information (translation, rotation, scale) that a free transform system would reference to move that entity's display component. For example, in Flow Jam, the fluid in the pipes would have a display component that references their sprite, a transform component that determines where that sprite is placed on screen, and a width component (narrow, wide) that would be checked against the constraint graph.

The state machine is another element of the game loop class. Its responsibility is to ensure clean transitions between different states of the game (ie, the main menu and the puzzle-solving gameplay would be two separate states). This means cleaning up the resources of the previous state and making sure the next state has the resources it needs. It also passes information between transitioning states, if necessary, such as the level data when transitioning from the level select state to the level state.

The script root refers to the field in the game loop class that is the root of the script tree. The game loop calls update on the root of the script tree, which then calls update on its children, and so on. These scripts define the mechanics of the game. The player input scripts process player input such as keyboard, mouse, or touch events and pass this information along to affect the game state (usually in the form of events). The behavior of clicking on the pipes to change their width would live here. The systems, as mentioned above, deal with the behaviors the components are meant to express, such as the free transform system that updates the screen position of any objects that have both a display & a transform component. The behavioral scripts handle the rest of the game's behavior; for example, the behavior of collecting the secondary objectives mentioned above lives here.

Challenges and Risks

Our most immediate concern involved updating the existing game to a playable state that is open for future development. With Flash support being discontinued and its already limited uses being dropped from modern browsers, adapting the codebase to a more accessible system was both a top priority and one that required an indeterminate amount of work that only became clear once the porting was underway. This process also required becoming familiar with both the ActionScript3 and Haxe languages, which many of us were new and as of yet unskilled with. The payoff, however, will be a port with stable support and cross platform accessibility. This in turn both makes the audiences that are necessary for tenable crowdsourcing more reachable and provides a welcoming development environment for future contributors.

This proved to be a challenge in the course of this project. Our original plan was to port two verification games (Pipe Jam and Flow Jam) from AS3 to Haxe, but due to the complexity of one of the versions and the technical dept present in both games, we chose to focus our efforts on only one of them, in the interest of making it a fun and enjoyable game. This was a hard decision to make because of the effort we had already put into porting both versions, but in the interest of producing a viable and well running game, we chose to discard Pipe Jam and more forward with the more developed Flow Jam.

Another challenge is that the codebase we were given for the game was clearly in a prototype state - even after we finished porting and were able to get the game up and running, there were significant issues with its comprehensibility, extensibility, and maintainability. Significant architectural issues were pervasive throughout the codebase that we refactored, leading to us taking steps to transition the codebase to a more traditional entity-component-system architectural pattern.

Adapting the existing code to fit into a new architecture proved to be a greater challenge than we anticipated. The original codebase had low cohesion, high coupling, and poor documentation, which combined with most of the team members being unfamiliar with an entity-component-system design, made the restructuring slower and more error-prone than anticipated.

After the game is available and open to be played, we will need to remain cognizant of the problems involved in ensuring that the game is entertaining enough to draw a crowd of users. Our collective experience in regard to game design is limited and will require consistent testing, feedback, and iteration to ensure our end product is enjoyable. This will require us to evaluate which particular game elements players respond best to. Once this is known, we can expand on those game aspects, improving existing ones or adding missing desired features. For example, if players are interested in a competitive dynamic, a comparison among the players' friends of who is completing the most levels or using the least buzzsaws could add to continued interest. There could also be a design space for a tertiary game loop with goals such as medals or some form of progression through the "world" of Flow Jam. Nonetheless, overcoming this final obstacle will see the utilization of crowdsourcing become a reality.

Initial Results

Our original plan of porting two games, as mentioned in the Challenges and Risks sections, proved to be too much for our team to complete in the project timeframe. Our new plan is to forego adding additional features and instead focus on a thorough evaluation and refactoring of Flow Jam's codebase. We believe this is a necessary task of greater importance than any feature addition could be. Owing to our awareness of how few of the existing verification games are in a runnable state, let alone a developable one, it would be prudent for the future of the Flow Jam project if we were to concentrate on reducing technical debt and bringing it to a future-proofed state. The current structure of the codebase features a large amount of coupling, which in addition to making parsing and debugging extremely difficult, also reduces any ability to hook in additional features. We have created a visualization of this structure by representing the codebase as a directed graph where each class is a node and an edge represents a dependency on another class[12]. The size of a node is relatively proportional to the number of lines of code in the class it represents. We have also displayed the number of 3-cycles in the dependency graph as a simple measure of coupling, since a non-hierarchical codebase is more difficult to comprehend.

We identified three classes that we believe to represent the worst cases of high coupling: Level, World, and GridViewPanel. We picked these because they are large classes with no clear, single purpose and are tightly coupled with each other, so we cannot concisely describe their functionality. Figs. 5, 6, and 7 below demonstrate the coupling of these classes and also that they form a 3-cycle of references, which further confuses the specific purpose they serve.

We started refactoring these classes by first comprehending their role and relationship to other parts of the codebase and then separating their functionality into components that fit within our ECS architectural plan. For example, Level and World should be refactored into classes that contain just data and no display properties. Furthermore, all input handling and displaying they do should be refactored into scripts or widgets as described in the architecture section.

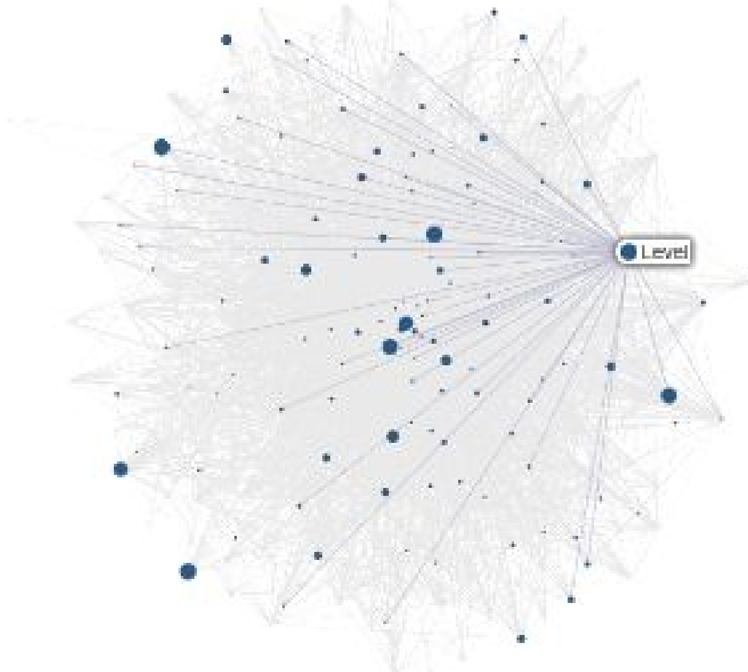


Figure 5: dependency graph highlighting Level

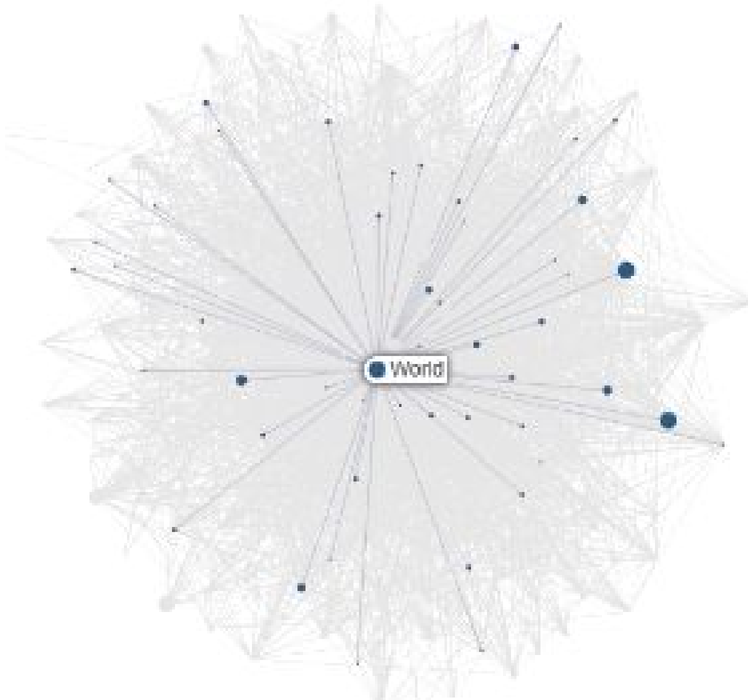


Figure 6: dependency graph highlighting World

World, in its original form was handling so many game components that it did not make sense in its current state. World as a module could not be described in any quick meaningful way. It did many things that should have been handled by the “game” itself. Registering achievements, user input, displaying the world and level were all part of the class. To refactor this class we moved all behaviors that had to do with any events that occur during the running of the game to scripts that are handled by the game engine. This involved checking that the behaviors were indeed unique to world and would not have side effects in other classes once the refactoring was done. After these initial changes were done World was already down to less than half its original amount of code. Next we looked deeper into some behaviors that were tied to what we want world to be, a place to keep the data related to the current game, and with its former purpose, a catch all class for things related to the game as a whole. These behaviors were harder to take out as many of them happen inside of methods that primarily deal with the world in its data storing role.

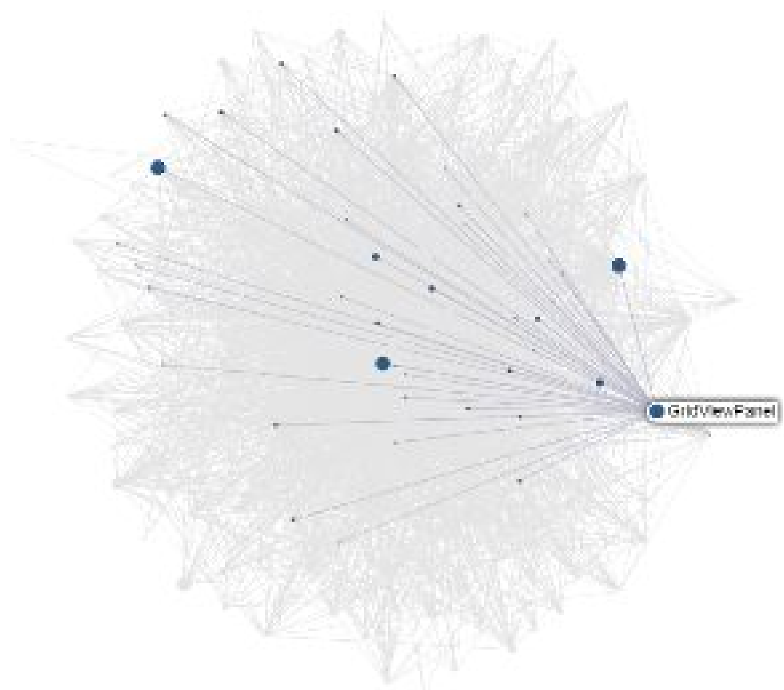


Figure 7: dependency graph highlighting GridViewPanel

Tied to the World module was the GridViewPanel module, which World utilized for broad display manipulation, such as camera panning and zoom. GridViewPanel was also host to a collection of ancillary miscellaneous functions, such as fanfare particle effects, tutorial tooltips, keyboard event handling. Separating this class involved several steps, the first of which was delegating its event handlers to the new centralized system used by all refactored classes. Following this organization, remaining functions were examined in the context of what GridViewPanel was intended to do, namely, provide a main display structure for the game world, and then pared down to only the essential relevant pieces. Unrelated behaviors were separated into new single purposed modules.

Future Work

Though we have taken the first major steps toward cleaning up the codebase, there is still work to be done. Much of the top-level architecture has been restructured, but many improvements can still be made for some of the lower-level classes. For example, most of the game objects extend a ToolTippableSprite class but this functionality should be moved out into a system that is responsible for adding tooltip components to existing entities. Most other classes would also be well served to be refactored in the ways Level, World, and GridViewPanel were, separating kludged modules and transferring state and event handling to the new game engine systems.

The next step after reducing the technical debt of the codebase and ensuring that there is a playable level is to playtest the game in its current state. This will involve asking people to play the game while being observed and then to complete a short questionnaire afterwards in order to obtain feedback. This data will then be used to evaluate and iterate on new features. A fleshed-out plan for playtesting can be found in Appendix B.

References

1. **Werner Dietl Stephanie, Dietzel, Michael D. Ernst, Nathaniel Mote Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popović. Verification Games: Making Verification Fun.** https://homes.cs.washington.edu/~zoran/verigames-ftfjp2012.pdf, **2012**.
2. **Thomas D. LaToza and André van der Hoek. Crowdsourcing in Software Engineering: Models, Opportunities, and Challenges.** https://pdfs.semanticscholar.org/787c/e8e0973fd21051260385bb27d74f6f53c7f9.pdf, **2016**
3. **Gleick, James. "A Bug and a Crash."** *A Bug and a Crash by James Gleick*, New York Times, **1996,** www.around.com/ariane.html.
4. **Mackie, Kurt. "Adobe and Browser Makers Announce the End of Flash."** *Redmond Magazine* 25 Jul. 2017. Web. 4/5/2018 https://redmondmag.com/articles/2017/07/25/adobe-ending-flash-support.aspx
5. **Erik Andersen, Yun-En Liu, Richard Snider, Roy Szeto, Seth Cooper, and Zoran Popović. On the Harmfulness of Secondary Game Objectives.** http://grail.cs.washington.edu/projects/game-abtesting/fdg2011/fdg2011.pdf, 2011
6. **Hunicke, Robin, Marc LeBlanc, and Robert Zubek. "MDA: A formal approach to game design and game research."** *Proceedings of the AAAI Workshop on Challenges in Game AI*. Vol. 4. No. 1. AAAI Press San Jose, CA, **2004**.
7. **Owens, Thomas. "Is it reasonable to build applications (not games) using a component-entity-system architecture?"** *StackExchange* 4 Feb. 2018. Web. 4/12/2018 https://softwareengineering.stackexchange.com/questions/186696/is-it-reasonable-to-build-applications-not-games-using-a-component-entity-syst
8. **Haxe - The Cross-platform Toolkit.** (n.d.). Retrieved from https://haxe.org/
9. **HaxeFoundation**. HaxeFoundation/as3hx. Retrieved from https://github.com/HaxeFoundation/as3hx
10. **Crowd-sourced Formal Verification Program Generates Thousands of Software Annotations.** (n.d.). Retrieved May 10, 2018, from https://www.darpa.mil/news-events/2015-05-28
11. **Verigames: Free puzzle games featuring logic, math, and science.** Retrieved May 24, 2018, from http://www.verigames.com/
12. **Flow Jam Coupling Graph,** https://verigames.github.io/flowjam/
13. **Foldit: Solve Puzzles for Science.** Retrieved May 17, 2018, from https://fold.it/portal/
14. **Mozak**. Retrieved May 17, 2018, from https://www.mozak.science/landing
15. **Paradox | Center for Game Science.** Retrieved May 24, 2018, from http://paradox.centerforgamescience.org/paradox/Paradox.html
16. **The Checker Framework.** 1 May 2018. Retrieved May 24, 2018, from https://checkerframework.org/
17. Stephens, Tim, and Dina Basin. **"UCSC and SRI Release New Game for DARPA Crowd-Sourced Software Verification Program."** *UC Santa Cruz News*, 23 June 2015, news.ucsc.edu/2015/06/binary-fission.html.

18. Dean, Drew et al. “**Lessons Learned in Game Development for Crowdsourced Software Formal Verification**.” https://pdfs.semanticscholar.org/57f4/06f355daae96c32f5deae0f9381c53147dac.pdf

Appendix A: Timeline

Old Timeline

Week 3

Resolve compiler errors for the Haxe port of Pipe Jam and Flow Jam.

Week 4

Continue resolving compiler errors & replacing external dependencies.

Week 5

Continue resolving compiler errors & replacing external dependencies.

Week 6

Continue resolving compiler errors & replacing external dependencies.

Week 7 (Midterm) Begin Evaluation Phase

Conduct playtesting

Begin work on Level.hx refactor

Week 8

Have implemented & tested the secondary objective of coin collection.

Week 9

Iterate & playtest the secondary objectives.

Complete Level.hx refactor

Week 10

Have implemented & tested client-side high-score tables on a per-level basis.

Week 11 (Final)

Have a release-ready build, fully tested.

New Timeline

Week 7(MIdterm)

Begin refactor of Level, World, and GridViewPanel haxe files

Evaluate dependencies of Flow Jam

Continue work on port of Flow Jam focusing on getting the program to a runnable state

Week 8

Use generated dependencies to guide refactor process

Continue port of Flow Jam

Week 9

Continue refactor

User test Flow Jam and assess results

Week 10

Continue Refactor and evaluate cohesion with generated graph

Begin implementation of new game feature

Week 11

Complete refactor showing that dependencies have been reduced and Level, World, and GridViewPanal follow a more intuitive architecture

Implement new game feature and have all work accessible in repository.

Appendix B: Playtesting Plan

Flow Jam will be evaluated by playtesting the games on students at a large university. We will test each player for 5 minutes while monitoring their gameplay. The players will be given a laptop with the game loaded and at the opening screen and will be provided with a brief description of what the game is and what they are doing in terms of verification while they play. They will then be given the task to play the game with no further instructions. During the test observers from our team will collect data on different metrics these will include:

1. Whether playtester ended playing early or not and by how many minutes
    a. We believe this will point to the version lacking in fun mechanics
2. Whether or not playtesters show of signs of frustration playtesters display. Signs of frustration will be defined as exasperated noises or stressed body signals
    a. This is a subjective measurement but it can be useful to gather this information in the instance were the tester does not offer information on their frustration freely. If we think someone was frustrated with the game we can ask them why and in what point of their gameplay did they feel that way. This will help identify areas where things may need to be changed or explained in a more precise way.

After playing the game for 5 minutes (or whenever the tester ends early) the playtesters will be asked the following questions about their experience playing Flow Jam.

1. On a scale of <u>very likely</u>, <u>likely</u>, <u>no opinion</u>, <u>not likely</u>, <u>never</u>, where would you place yourself in wanting to play the game again?
    a. This will be a general measurement on entertainment value of the game.
2. Please rate the level of frustration you experienced while playing the game on a scale of 1 to 10.
    a. We believe frustration is a good measurement for assessing the game. Players who experience this are less likely to want to play the game in the future. Frustration could have many causes when playing a game and as we will not be changing any major gameplay mechanics of the game we will want to measure this in some way.
3. Are there any aspects of the game that were unclear or was there any point when playing that you didn't know what to do?
    a. This will be used to further evaluate the games entertainment if the more general question yield no useful information.
4. Would the addition of any of the following features make you change your answers to any of the previous two questions? Leaderboards? Badges/Achievements?
    a. This question will help us get more input for features that we want to add to the game to increase player engagement.

After an initial round of playtesting we will use playtesting sessions of this structure to incorporate additional features into the game. As game improvements are proposed and implemented, these playtests can be repeated using an A/B testing format wherein testers are randomly assigned to either play a control version of the game with no changes or a modified version that has a new feature, such as secondary objectives or a different scoring system. Using the measures of player engagement and enjoyment (based on observation and the self-reported feedback) we can determine the efficacy of the feature as a gameplay enhancer and whether it improves the game.