Fun Factory

Connor Levesque
Middlebury College
Middlebury, VT
clevesque@middlebury.edu

Max Shashoua
Middlebury College
Middlebury, VT
mshashoua@middlebury.edu

ABSTRACT

Fun Factory is a mobile puzzle game in which the player designs a factory to solve a host of problems. By creating an assembly line to move, spin, paint, and weld crates together, the player combines individually simple parts to solve complex problems. Fun Factory was built over three months and forced its developers to tackle difficult problems in the domains of software architecture, game design, and algorithms. The final product represents a functional game prototype and attests to the strength of its core gameplay with 8 distinct machines and 15 playable levels. Although Fun Factory is not yet ready for the mobile storefront, it is only separated from publishing by visual polish and the inclusion of additional levels.

1. INTRODUCTION

When faced with the problem of choosing a project for our Senior Seminar in Computer Science, we knew immediately that we wanted to create a game. Games are a medium we are both passionate about, and this project represented an opportunity to level up our software development skills and produce a product worthy of the resume. In deciding what sort of game we would implement there were several factors to consider. First was efficiency. The game needed to be implemented by two developers over a three month period while still providing deep and satisfying gameplay. This led us to design a "problem-solving" game where the player must create a solution to a problem using a set of tools. Like a typical puzzle game, Fun Factory's levels are small and discreet, but because there are many solutions to a given problem, the experience of an individual player does not need to be tightly managed. Thus the game relies more on the player using their knowledge and developing facility with the tools provided than on "gotcha" moments common in puzzle games.

Fun Factory delivers on this mission by charging the player with the task of designing an assembly line for each level. Each level has an output which the assembly line must produce. This output is composed of crates which are produced be crate generators placed at every level. The players goal is then to move, weld, spin and even paint these crates to produce the desired result. As the complexity of the goals increase and new machines like conveyor belts, spinners, and welders are given to the player, the levels become progressively more difficult. This allows for a wide range of levels and experiences to be crated because each solution is built from the same simple elements, which are often combined in unexpected ways. Early on, for example, the player is tasked with using Pushers, Sensors, and Wires to branch one stream of crates into two. Although an early level only requires the player to send the crates toward two separate drop zones, later levels require branching multiple times and recombining the components into a larger shape. In this way Fun Factory incrementally advances a players knowledge with overwhelming them all at once. The final product includes 8 machines, 15 levels and many hours of gameplay.

In the paper that follows we discuss similar games which inspired Fun Factory, our implementation process, the architectural and game design choices we made, and possibilities for future work.

2. RELATED WORK

With regard to the game's overall structure, we were heavily inspired by Infinifactory, a game produced by the independent developer Zachary Barth. In large part Fun Factory could be characterized as Infinifactory 2D, adapting many of the game's core elements for the mobile platform and thus a larger audience. Like Infinifactory, Fun Factory focuses on manipulating simple game elements to solve complex problems with many solutions, making both games less about puzzle solving then they are about problem solving. Like Fun Factory, Infinifactory was also made using the Unity game engine. [1]

The design of Fun Factory was also influenced by Enigmo, a 2D puzzle game created by Pangea Software. Initially released in 2003 for PCs, Enigmo was released on iOS in 2008, and was voted one of the top mobile games of that year. The objective of each level of Enigmo is to guide drops of liquid around and through obstacles and into specified barrels using the tools and machines available. The most influential aspect of Enigmo was the level design. With only a few small changes to a level, the designer can drastically change the complexity of a problem, changing the level from easy to extremely difficult. [2]

3. METHODS

3.1 Unity

The principle tool used to create Fun Factory was the Unity game engine. A game engine is a platform for creating games with built in functionality for features common to games such as lighting, physics, the need to display content to the designer, and compiling to executables. Unity is notable for being the most accessible game engine which is nonetheless used to build several major titles and is hugely popular among independent developers. Although Unity does not support cutting edge triple-A graphics, it can easily support artists who are not in pursuit of photorealism. Without a platform like Unity, creating Fun Factory would simply not have been possible.

3.2 Implementation Steps

To implement Fun Factory, we pursued a two step process which allowed us to implement the core of the game before concerning ourselves with the creation of a user interface. Playing a level in Fun Factory consists of placing machines to create an assembly line and pressing the run button to start the assembly line. Conveniently, the process of designing a Fun Factory level consists of the same to steps, placing machines in the scene, and pressing Unity's run button to play the game. This meant that we could effectively play our game using Unity's user interface. By setting our factory to always run by default, we could use Unity's run/pause buttons to test the core operation of our factory before any actual UI elements were created. Not only did this cleanly separate the development of the game's features from the development of its UI, but it allowed us to put off UI decisions until much of the gameplay we were designing the UI for was already in place.

3.3 Architecture

In architecting our project, we pursued a strategy roughly similar to the Model View Controller (MVC) pattern common in software development. Although we did not explicitly label classes as part of the View or Controller, we did create a distinct model to separate the game state from its display. At the core of our model where the GridOf<T> classes which tracked the position objects in a 2-dimensional array. The use of a generic class in this case allowed us to write one class governing grid behavior and use it to track the position of machines in a GridOf<Machine> and crates in a GridOf<Crate> aliased as Machines and Crates respectively. This distinction is important as in many cases a crate and a machine occupy the same position, for example if a crate is sitting on a conveyor belt. Machine and Crate were also key classes in our model, governing the state and behavior of their respective objects in the scene. They also were part of the game's deepest class hierarchy where both inherit from the GridThing class (which includes object positions) with Machine becoming the superclass for each of the game's machines (Conveyor, Rotator, Pusher, etc...). In hindsight, a refactor which would improve the game would be to extend this hierarchy further to distinguish between machines that can be placed by the user and those that cannot such as generators and painters. This would have made some code significantly cleaner with respect to manipulating placed machines.

Returning to the GridOf classes "Machines" and "Crates", it is notable that both of these classes are singletons, i.e. instances of the singleton design pattern. In the singleton pattern a class carries a single static instance of itself which is accessed by static methods on the class. The result of this is twofold. First, there is effectively only one instance of the class as all relevant methods point to this single static instance (hence singleton). This is appropriate for our Grids, GameManager and LevelManager as multiple instances of these classes would always lead to bugs and confusion. The second feature of the singleton pattern is that by exposing all its methods statically, the class is effectively global and can be accessed anywhere in the application. This allowed us to contact the Grids for moving and removing crates and machines, or the LevelManager to load a new scene, from anywhere in the application. For us this was a significant convenience by which we avoided passing Grids through long method chains or creating a reference to them in almost every class. However, just as global variables are to be discouraged, global classes in most software development should be discouraged as well. When many developers work on a project for multiple years, there is the very real concern that one of them will abuse the global nature of the singleton class, for example, by introducing code governing audio into the game's physics engine or vice versa. For this reason the singleton is often rejected as an "antipattern" which leads to poor software development in the future. However, in the case of our project, spanning two developers and three months, the liberal use of singletons made our code easier to both read and write.

One of the design patterns we reused throughout the game was our grouping protocol, specifically when grouping Wires, DropZones, and Crates. We designed this protocol to be efficient and reliable. We gave each class a reference to its group, and each group references to all its group members. This class structure made it easy to apply our protocol to all the objects that shared this grouping property. A future refactor could be to distinguish objects between grouping and non-grouping objects higher up in the class hierarchy. To create a wire group, we take one wire and look for any adjacent wires. If we find any, we then merge their groups. This ensures that all connected wires will be part of the same group. This protocol was identical for DropZones, and only differed with Crates because of the interaction between merging CrateGroups and Welder placement.

3.4 Game Design Decisions

The driving motivation of Fun Factory's design was to create a deep gameplay experience from a minimal number of features. In some sense this is always a pillar of game design?to build more from less?but with one semester and two developers to work with, efficiency is at a premium. Fun Factory embraced this goal because at it's core, Fun Factory is about phrasing problems and giving users the tools to solve them. Rather than creating large game scenes for the player to explore, designing a Fun Factory level consists of placing a generator to produce crates, a drop zone of the desired shape to establish a goal for the player, and selecting which machines the player may use to complete the level. This meant that level design, which is often labor intensive, claimed only a small part of Fun Factory's total development time.

Furthermore, the individual features implemented in Fun Factory are all relatively simple. Each machine is charged with no more than manipulating crates or other nearby machines. However, by allowing the player to combine these features in novel ways, deep and complicated scenarios emerge naturally from the game's rules. In fact the most difficult part of Fun Factory's implementation arose from the need to resolve edge cases which emerged naturally from the players placement of machines. Because the player could design any factory, it was important that we resolve a host of complicated scenarios. What should happen if two crates want to move into the same square, or if two conveyor belts push a group of crates in two different directions? Where possible we resolved these situations intuitively, but in many cases it was less important how an edge case was resolved as long as it was resolved and thus could not interrupt gameplay.

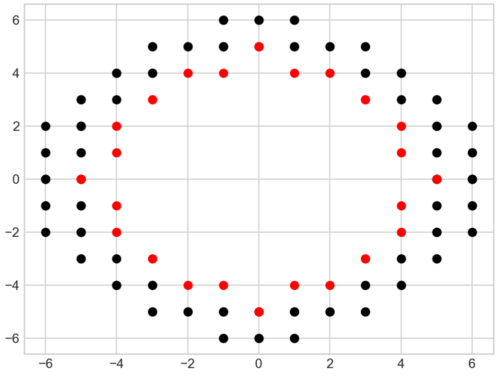
3.5 Rotation Algorithms

Before rotating any crates, we needed to check for obstacles in their path. Given a point p and a center of rotation c, how should we choose which points to check along the circle?

The midpoint circle algorithm is used in computer graphics for efficiently rasterizing circles. We initially chose this algorithm because it is very fast?completely avoiding the use of trigonometric functions. However, after implementing and testing this algorithm, we determined that we needed thicker selections than what this algorithm was giving us, so we abandoned it to create our own.

Our objective in creating our own algorithm was to address the 'thinness' problem of the midpoint circle algorithm. The concept behind it is simple: using small angular increments, we tick around a circle with center c that goes through point p, and select squares using small thresholds in distance from neighboring points. The selection process is based on a square centered around the nearest integer point with 9 sub-squares (see Figure 1). After completing one full circle, we remove duplicates from this list and return an in-order list of points along this new thicker circle. We found that this performed much better than the midpoint circle algorithm, grabbing a thicker and more intuitive set of grid squares to check.

Figure 1: Here the red dots denote the points given by the midpoint circle algorithm where all points denote those given by our custom algorithm.



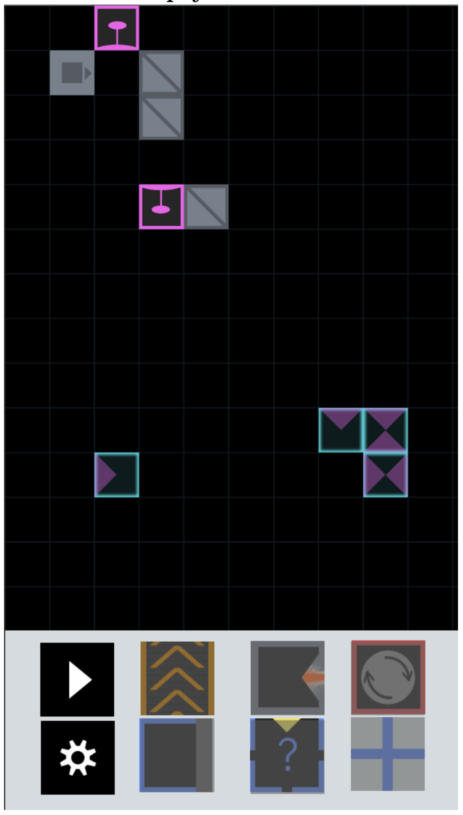
3.6 Github

Our project can be viewed and cloned at https://github.com/connorlevesque/FunFactory.

4. RESULTS

After three months of work split between the two developers we created 8 distinct machines to be used by the player to solve 15 levels and several hours of gameplay. This also included the creation of a user interface allowing the player to design assembly lines as well as start and level select screens to navigate between the levels. The following images show a Fun Factory level in action and are intended to give a sense of the game and its features (see Figures 2-5).

Figure 2: An empty level in its initial state.

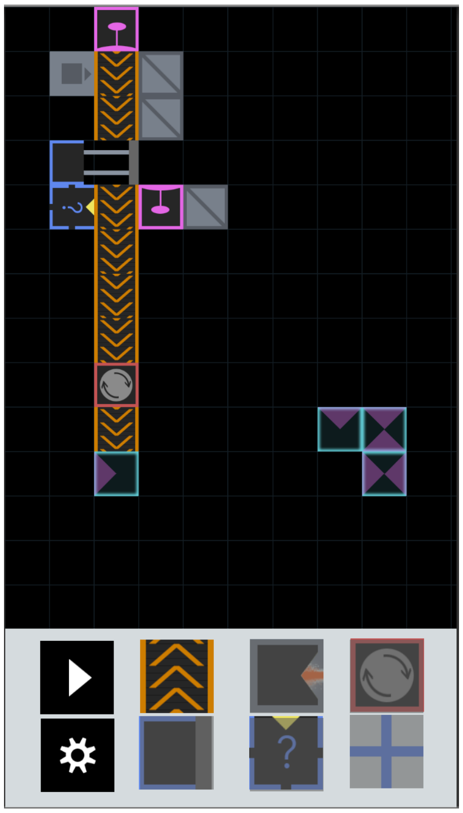


5. DISCUSSION

Beyond the final product, we also learned a great deal from the process of implementing Fun Factory. Our primary take away from the development process was the importance of careful planning in software development. During the early stages of Fun Factory's development, a significant amount of time was dedicated to planning our class hierarchy and anticipating edge cases before any of the associated code was written. The importance of this planning phase was most prominent in the crafting of the game's update loop which governed the manipulation of crates from one update step to the next. This was a difficult problem to approach because every edge case must be handled smoothly. We ultimately settled on a two step process where machines first recorded the forces they intended to exert on crates and then theses forces were resolved to select one cardinal direction in which to move (crates in Fun Factory do not move diagonally). We also avoided a number of ambiguities by settling on an order of operations such that welding always occurs before rotation which always occurs before pushing of any sort. The forethought that went into the game's architecture contributed significantly to a smooth development process and allowed us to avoid a long a painful bug-fixing stage.

As for Fun Factory's weaknesses, the game's most significant drawback in its current state is its lack of visual fidelity. Although Fun Factory is engaging, the lack of professional artwork and sound effects makes the game less appealing

Figure 3: The first part of a player's solution to the level.



and also somewhat more difficult to understand. If each machine's artwork better communicated its function, it would be easier for new players to use them intuitively. The addition of audio would also be a significant improvement. The satisfying clink and whir of a factory in motion is currently absent from the gameplay experience. That said, improved artwork, sound affects, and additional levels are all that separate Fun Factory from final publishing on the App Store.

6. REFERENCES

[1] Zachary Barth. *Infinifactory*. 2015.

[2] Pangea Software. *Enigmo*. 2008.

Figure 4: A completed solution.

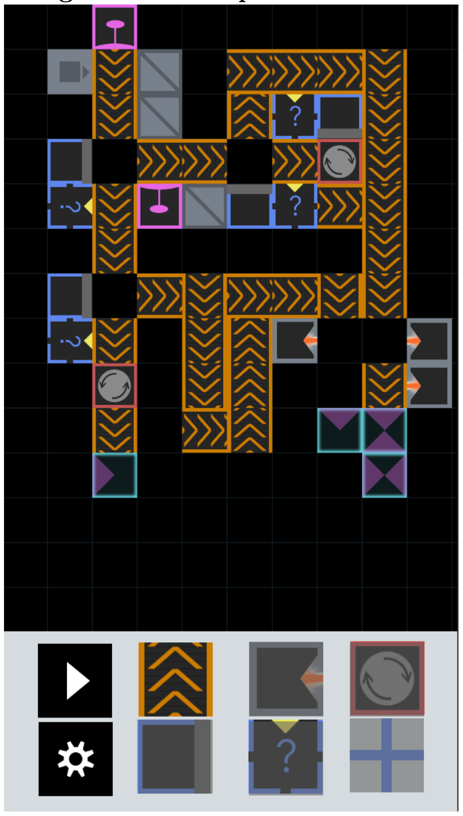


Figure 5: The running factory.