Fairytale Factory Fantastica

Implementation Summary

MUFFIN MAN STUDIOS

Spring 2014

Edited by Barrett Adair

Fairytale Factory Fantastica

Implementation Summary

Table of Contents

Team Members

Barrett Adair

Core Architecture, Networking, AI, Installer, Animations

David Domasig

AI, Audio, Help, Licensing, Optimization, Unit Testing

Russell Dillin

Networking, Graphics, Menus, Animations

Stephanie Nill

Intro Sequence, Menus, Animations

Elizabeth Phillips

Graphics, Storyboards, Menus, Intro Sequence, Animations

Framework and Tool Selection

Barrett:

We wrote our project in Qt (specifically, QtQuick 2). Qt is a cross-platform, event-driven C++ environment. We couldn't have been happier with our decision to use Qt. Qt extends C++ using "signals" and "slots" (similar to events and event handlers). Qt has its own networking library which is very easy to use. I hadn't taken networking, but the library was still easy for me to use. We made use of Qt's excellent thread library as well. One of the best selling points for Qt is its namesake declarative markup language, QML. QML interfaces seamlessly with both C++ and Javascript. QML enables you to abstract graphical elements of your project away from C++, and is also easy to learn, easy to write quickly, and easy to read. Everyone on our team enjoyed using QML. Most animations take a just a few seconds to write. Particle effects are amazingly simple to use. The documentation online is extensive.

The most difficult part about using Qt is the initial learning curve. Coding event-driven C++ feels very strange at first, and Qt's way of doing things is going to take some time to learn. If you use Qt for your project, I recommend making a rapid prototype in the first week of class. It's a lot to learn, but you can definitely do it in a week. I recommend that one person on your team, preferably the leader, should be in charge of coding the first prototype. He should then rewrite it at least once, learning from inevitable mistakes in prototype design.

Qt extends the language with signals and slots. A special program parses your source code and generates a whole bunch of "normal" C++ code from the Qt code you write. Then everything is compiled with a C++ compiler of your choice. You also have to use a few "magical" macros. It's all pretty daunting at first, but it's totally worth learning. With so much happening in your code, you are going to encounter some confusing errors. Google is going to be your new best friend for most of these.

The QtCreator IDE is a work in progress, but I ended up really enjoying it. I will definitely use it in the future for any personal C++ development. The IDE gave us some issues at first, but it is improving with each release. I started using a new beta build halfway through the semester that had several bug fixes and new features. You don't have to use QtCreator to use Qt, but I highly recommend it.

In development, we compiled with MinGW 32bit and 64bit for Windows, and clang 64bit for OSX.

We used GitHub for our code repository. However, I can't stress enough to you about how bad the GitHub Git client is. If you use Git, don't use the GitHub client. But I don't recommend using Git at all. Git is very powerful, but it has a steep learning curve and is simply not the best tool for this project. If all of your team is using Windows machines, I highly recommend using Subversion with TortoiseSVN client. I used TortoiseSVN previously and it is by far the easiest version control software I have used. We wasted far too many hours on Git.

Russell:

We chose Qt primarily because we were most comfortable coding in C++. We knew about QML, but it wasn't a huge factor at the time of our decision. After working with it, I can say that QML is a real joy to use. It's high level, powerful, and a cinch to learn. Everyone on our team picked up QML and was able to contribute to the user interface.

The following is a small example of a rectangle which changes its color when clicked. It's a silly example, but it's a nice taste of QML syntax and how easy it is to create interactive objects with QML.

```
Rectangle {
    id: example_rec
    width: 400; height: 600
    z: parent.z + 1
    anchors.centerIn: parent
    color: "red"
    MouseArea {
        id: example_rec_mouseArea
        anchors.fill: example_rec
        onClicked: {
            example_rec.color = "blue"
        }
    }
}
```

Once we understood the power of QML we decided that our backend logic (game core, AI, networking) would be implemented in C++ while our GUI would be implemented in QML. That would create a good, clean separation between the two as well as allowing us to work quickly by leveraging the power of QML. I think it was a great decision.

QML makes animations quick and fun. That's all that there is to it. We were able to easily make fun, complex animations in the course of a single afternoon. In High Noon's 2013 implementation summary, they stated that, "Windows Presentation Foundation was not meant for intense graphic rendering or animation. Because of this, we had to create our own animation engine."

If nothing else, I would highly suggest working with Qt and QML for the amount of ease in creating animations. Qt allows you to create infinitely intricate and interactive animations. For example, you can have four PropertyAnimations happening sequentaily in a SequentailAnimation within a ParallelAnimation so that four other PropertyAnimations are happening within a SequentailAnimation. QML handles it all with ease. It was a lot of fun to laugh at unintentional animations while trying to code them. QML makes animations fun.

I also want to point out that we would have been totally lost without Qt's online documentation. Both the official qt website and the QtCreator IDE have excellent examples, of which we made great use. We constantly referenced them throughout the project. You will definitely want to use those when using Qt.

Be realistic and realize that there is only so much time to complete this project. Make a detailed anaylsis of your schedule for the semester and determine just how much time and energy you are prepared to spend. Now that you know just how much time you will have at your disposal, you must ask yourself a question. How do you want to spend the time that you have. Do you want to spend the majority of your time tackling a new tool? Everyone learns new tools and technologies during Capstone. That's part of the experience. You are going to have to push yourself and learn new things. But at the same time, you need to be productive and push the development and progress of the project. A balance must be met.

You can only attempt to learn so many new technologies at once before you begin to feel overwhelmed by the foreign, unfamiliar environment. For the most part, we chose the tools that we already knew. We wanted to get going quickly. We didn't want to learn another

programming language. We didn't know Qt at the time, but it was easier to choose Qt since we were familiar with C++.

Git is complicated. Nobody on our team knew Git very well, but we decided to use it anyway. There were many moments when we nearly switched over to Subversion, but we continued to use Git with Github and Github's official GUI programs for Windows and Mac. We got through just fine, and we all learned a thing or two about Git. It wasn't nearly as frustrating once we worked with Github's intended workflow. Submitting pull requests through the website rather than attempting to merge using the GUI clients seems to be the best way to combine branches.

Google Calendar became a great tool for our group. It's a simple way of committing to be in the lab at a certain time. We were able to communicate with each other when we were able to work. It really helped our group. Open communication is essential for this project. I'm thankful that our group was willing to keep a detailed work schedule as well as communicate which tasks we were working on at a given time.

Stephanie:

When our team chose to use the Qt framework, we by default decided to use the Qt Creator IDE and all of the features associated with it. Our team was satisfied by this choice, especially when we realized that the IDE could be used on a Mac or a Windows platform. The IDE has a tool in it for developing the user interface called Design, but our group did not look into that very much. I would recommend that future groups spend time trying to figure out how to use that, as it could potentially simplify the design aspect and placing of graphics on the screen. Since Qt is still so new, it's IDE and the markup language that it uses, QML, had some bugs. Many of the objects and functionality that should have worked did not work for one reason or another. It was always possible to work around such problems or develop another solution, but that frequently lead to bulky and more brittle code.

GitHub was the tool we chose for storing our code repository. Since GitHub was a new tool for most of our team, there was quite a learning curve in the beginning. We overwrote some parts of our code several times, and often had to copy and paste code instead of being able to manage the code from the GitHub client or site. This lead to development delays, but was mainly due to our inexperience with the tool. The GitHub client for the Mac platform was not as developed as the one for Windows and did give us more trouble than the other. Once we learned better how to work with GitHub, we were able to utilize it to

our team's advantage. The strategy that we found worked best was to create new branches any time we decided to merge branches, and that way we could test if our merge was successful while leaving both original copies of the code intact.

Theme Selection

David, Elizabeth, Russell, Stephanie, Barrett

We spent a lot of time and thought on the game theme. We had at least three meetings, each about an hour long, that were dedicated to finding a theme and brainstorming about it. These meetings helped set the tone for the rest of our project. Theme selection can make or break a game during user testing, so we needed something that would be enjoyable and relatable for all age groups. We needed a game theme to account for a game board divided into four rotating sections. Selection was done democratically, and many ideas were presented. During brainstorming, there was no such thing as a "bad idea." In the end, our team was able to narrow the choices down until we all agreed on the "Hansel and Gretel in a candy factory" theme. Everyone is at least vaguely familiar with the story of Hansel and Gretel. We decided to have them in a factory that produces gingerbread houses, hence the gingerbread squares and gumdrops. The theme allowed for whimsical and simplistic graphics, bright colors in contrast with drab machinery, wacky animations, and a humorous storyline.

The theme of the project needs to be hammered out early on in the design process. Allow for plenty of brainstorming time, but do not let theme selection be a time drain in the development process. Be sure to choose a theme that a wide audience might enjoy. As soon as the theme was selected, our team enjoyed an instant boost of morale. The weekend following theme selection was one of our most productive weekends of the semester.

Our development of how the theme would play out was done with a little less forethought than the theme had been. The theme of Hansel and Gretel at a candy factory more fully developed as the project continued. We had a fairly good idea of the game screen visuals, and we always wanted elaborate animations. Once the theme was decided, Elizabeth made some storyboards and we discussed them. Little else was done regarding the theme until after the UIP. While this definitely put us ahead of schedule with the game functionality, it also left a lot of questions about how the final game screen visuals that remained until after spring break.

Product Architecture

Barrett:

The defining feature of our architecture is the separation of the QML/Javscript frontend from C++ backend. The C++ class that ties the two together is the GuiGameController. The GuiGameController implements a GameCore. The GuiGameController also has 3 AI player objects (i.e. a FinalPlayer object). It also has a NetworkInterface object.

The GuiGameController runs on a dedicated thread (the QThread declared in main), which in turn is the thread that its AI and NetworkInterface objects run on. This prevents the main application from locking up while the AI is processing and keeps all the interactive graphical elements responsive. Strangely, Qt's QQmlEngine cannot communicate directly across threads unlike most QObjects, so you will find in the GuiGameController header a Proxy class that passes messages back and forth between threads. (The Proxy class was added late in development, and obfuscates the code enough to warrant a mention here.)

We knew we would need to make our Board representation run as fast as possible, since our AI would make heavy use of it, so we decided to represent our boards as 64 bit integers. David did most of the work on this front. It's a lot of bit twiddling, so it's difficult code to understand, but it does run very fast. (Our release build was built using unofficial 64 bit builds of both MinGW and Qt 5.2.1 dynamic libraries, which greatly increased our AI speed by compiling to use 64 bit registers).

The GuiGameController is the where most of the Qt-specific C++ code lies. Most of the rest of our code is standard C++11, which made writing console test drivers for the AI easy.

The C++ architecture was the result a rapid prototype followed by two redesigns. I'm very glad I took the time to redesign. The end result was modular and straightforward. We changed very little about the C++ architecture after spring break.

User Interface Design

Stephanie:

Since we were using Qt's framework for our game, we also used their markup language for our graphical user interface. Their markup language is called QML and is very much a combination of HTML-type markup combined with functionality reminiscent of JavaScript and PHP. QML is not hard to learn how to use, but it is so broad in the amount of functionality that it offers that it was hard to master. The Qt project's official site was a great resource for us. They provided documentation and descriptions for each element and all the functionality that it offered, and they also usually provided helpful examples of how to use them. The support on the online forums was more limited due to the fact that Qt seems to have originally be mostly used for developing mobile applications and the support for those was different than for desktop applications.

We made QML work for us, but our QML files became very cumbersome and bulky, as did the design of our user interface. This was due to the fact that we did not "design" the interface so much as it "grew" along with our project. I would highly recommend against this approach, as it makes the code hard to follow and hard to fix. However, the QML was very easy to use, and the way it interacted with the C++ code was very elegant. Qt uses a system of signals and slots to send information back and forth between the C++ core and the QML front-end. This is how we got the values to display on the game board and also how we kept the game core up-to-date with the current moves made by the users via the GUI.

We had a main QML file that connected with the rest of the game, and then the game screens each had their own QML file. An instance of each screen was placed in the main QML file and was hidden using that property within the QML element until it was necessary to show it depending on where the user was within the game. Making each screen visible was easily done by changing the visibility property of the element. All buttons and functionality on each screen were also disabled until the screens were made visible. Each screen had its own QML file, and within it, we placed all of the graphics that our graphics lead had developed. It worked best to get the graphics about the size you wanted them before placing them within the game, but they were also easy to scale down within the QML itself. We tried to get each graphic separately so that we could place and animate them independently. We usually did not give each individual graphic and component their own separate QML file, preferring instead to encode it within the screen that it was to be used, unless it appeared multiple times throughout our game or if it had complex and lengthy logic.

Each QML file created a new element of some type, and these elements could be manipulated using their properties or by using JavaScript functionality, such as OnClicked() event handlers, or by creating original functions. Functionality could be accessed across files by making a Connection in one file with a function from another file. This allowed more reusability and flexibility for each individual component.

Transitions and animations could be easily done by using the built-in functionality offered by the QML. QML offers ParallelAnimation, RotationAnimation, SequentialAnimation, and much more. These, when called, perform an animation on some object, and the animation itself can be customized and tweaked using the properties inherent to each of the animations. Depending on the animation desired, we often just used these animations and changed the position of a particular graphic onscreen. More often than not, however, we frequently had to make sprite sheets for animations that involved any changing of the actual form of the graphic. Once the sprite was made, the animation was easily done using the QML SpriteAnimation, which was also able to be customized.

Placement was rather tricky using the QML. Most of the time, our team tried to stay relative by placing graphics relative to each other. This worked best and is what we recommend. However, some pieces had to be placed by absolute X and Y value to better work with the animations, which we later discovered could still be placed using the same anchor system that kept the rest of the pieces relative to each other. The anchor system is the most flexible way to place objects on the screen in order to have more options when trying to play on other resolutions.

Estimating the amount of time spent on our user interface is nearly impossible. We had at least two people working on it from the beginning, but then for more than half of the time, most of the team spent at least some time working on different parts of it. It is best to start it early and never quit working on it. It is the part that the users will see the most, and it is also the part that differentiates your game from every other team's game. It is important to make it feel coherent and innovative.

Elizabeth:

User Interface is extremely important in designing a game; it needs to look pleasing to the eye while maintaining user friendliness. We needed a simple interface that was easy to use and would allow for animations that give the user a fun and immersive game experience. We planned from the beginning to focus heavily on animations.

Russell:

I wanted a simple user interface that anyone would be able to navigate. I looked to simple user interfaces from popular games for inspiration. Popular iPad and tablet games such as

Angry Birds and Badland have very simple user interfaces. Due to the fact that there is limited screen space on tablets and smartphones; these games have clean, well-designed interfaces. I thought it would be best for our project to model these interfaces. I wanted menu navigation to be straightforward, and I wanted users to easily understand how to play our game. As the semester progressed, the design of our user interface underwent many changes. However, the concept of simplicity remained a top priority throughout development.

One of the most important early decisions we made regarding the user interface was to omit the story mode/user profile features, even though many teams before us included these features in their projects. The decision was a bit of a gamble since we knew the other teams planned to have such features, but we decided that since users would be testing for only a few minutes, implementing a story mode wouldn't be worth the effort. We reasoned that a campaign mode and user profiles would take too much time to program and would introduce too many edge cases into the code, which would have greatly slowed development and debugging, while leaving only a small impression on our users. We decided instead to pour our GUI development efforts into spectacular animations and visual flair.

Because we decided not to have a story-mode, it made the most sense for us to reveal the storyline through the user interface. We have an introduction video which leads into our title screen. The title screen introduces a setting and a sense of atmosphere. The buttons and menu selectors became a part of the world to have a feeling of immersion.

Barrett:

The QML is not nearly as neat as the C++. All 5 of us spent lots of time in the QML. There are, unfortunately, quite a few of global variables in Main.qml. As you might expect, adding features was easy (because the variables were always available), but debugging them was not. If we were to write the QML again, we would replace a lot of the imperative Javascript code (reassigning variables, checking Booleans, etc) with a more event driven, declarative approach. For instance, instead of changing a Boolean to toggle sound, we could have a *state* on a sound effect that is dependent on the *state* of the toggleSoundButton.

The board you see on the screen gets its pieces from the C++ GuiGameController, which is "in charge" of maintaining the actual game state.

Also, note that the "Loading…" screen is rarely used for loading anything at all. It's mostly used to hide animations and let them finish running before leaving a certain screen.

GRAPHICS AND ANIMATION

Elizabeth:

Graphics are just as important as the programming. A good game will require lots of time and detail in the graphics. Many of our graphics had to be hand drawn from scratch. Others were collected from Google Image search for images with explicitly non-restrictive licenses. The challenge with using open source images was finding images that matched the game's theme and art direction. We had to set out a lot of time to dedicate to developing and drawing graphics for our game, because quality open source images that matched our project were very hard to find.

We ended up using over 200 images in our game. The graphics I developed were first sketched by hand. After going through several sketches and version finding which one looks best, I would scan the image into my computer and trace it on the ArtRage Studio program with a Wacom drawing tablet. From there I would import it into Photoshop to add color, detail, and texture to the image. One of the biggest challenges was creating images without having planned out in detail how they should look. Often, I would have to go back and make changes. I had to make several versions of the witch, Hansel, and Gretel for the various screens and animations. We did not completely plan out how the animation and graphics would work and fit together, so it caused us to have to go back and redraw certain images for certain conditions. Spending time in sketching and brain storming before jumping into designing the graphics is important, and is one thing I wish we might have done more of before jumping into the graphics. Over all, designing the graphics was a lot of fun. Being able to learn more about the process of graphic design and being able to share the load with team was great.

At first using a Wacom drawing tablet was difficult. it was rather difficult to draw straight lines and draw in detail with a Wacom. It was considerably easier to simply draw the image on paper, and then scan the image, and then trace the image using the Wacom. The Wacom was a big help overall.

QML animations are simple and easy to use. We have a lot of animations going on in every screen. Using QML animations was a lot simpler than using sprites. Most of the animations were easy to code. The only difficulties I had with animation was finding the correct y or x values on the screen. QML it seemed a bit overwhelming at first because of all the features, but it really did not take as long as I expected to learn. It's a really simple language. Overall,

it was a good feeling and a lot of fun to see the sheer volume of animations we implemented in our game, thanks to our framework.

Particles systems also played a huge rule in our user interface. Users will notice that we have particle effects on many of our screens. The smoke on the factory, the fog, the fire, and the glittery effect when you place a piece are examples of QML particle effects. Learning the particle system was at first a bit challenging, but it became a visual boost for our game and was a lot of fun to learn to use. Particles systems do use up a lot of CPU, so we had to keep them from running in the background when their parent screen is not in view. The particle effects helped our end user experience tremendously. They were totally worth the effort.

Russell:

I had some skills with Photoshop, but didn't have any experience creating graphics for a video game. I leveraged a lot of tutorials and help for the internet. I drew the claw sprites and cog sprites from scratch, frame-by-frame. It wasn't the fastest nor the most effiecent way of doing it, but he knew that it would result in a high-quality, smooth animation. He opened the claw by 1 pixel, saved it as a new image and continued until the claw was completely open. It was 75 frames in all. I had no experience writing scripts for Photoshop, but knew it was possible. A quick Google search led to a free, open source Photoshop spritesheet script hosted on Github (https://github.com/rohanliston/adobe-spritesheet-scripts/tree/master/photoshop_spritesheet_exporter). It may seem daunting to create graphics, but anyone can do it with a little time and practice. There's a massive amount of helpful information online. Don't be afraid to ask for help or search online. In the end, it's more productive to seek help than to waste time struggling to figure something out on your own.

Barrett:

I regretted using sprites. They are very inefficient in memory, take a long time to make, are hard to change, and are generally not suitable nor necessary for a game of this scale. We hit a wall with sprite performance at one point. I created an icing sprite with around 100 frames to show icing being spread onto the board, but it ran far too slowly, so we had to get rid of it. Sprites work by switching images really quickly. This is much less efficient than letting Qt animations move things around with OpenGL and other under-the-hood magic. The characters swinging and flailing before being thrown into the fire at the end is an example of an animation without sprites. Note: When using sprites in QML, keep in mind that the SpriteSequence element is more robust and flexible than the AnimatedSprite element.

Artificial Intelligence

David:

For our project it was a pretty straightforward decision to stick with C++ for the AI. Ideally, this would allow for greater efficiency and speed in our code. We created an abstract base class, Player, which each AI implementation would implement in order to maintain a consistent interface with the game controller. This design decision allowed us to implement any kind of AI algorithm while still being able to function with the rest of the program. Another key feature of our AI was the implementation of bitboards. This feature allowed us to represent the game board as a 64 bit integer, which allowed for rapid calculations and comparisons using bitwise operations.

The most important lesson learned from the architecture side of the AI is to not get offended easily. The AI lead is just a title, and sometimes swallowing one's pride is part of that position when a better algorithm beats yours. Be open to communicate with your teammates about improvements, enhancements, and, if needed, overhauls.

Ideally, the AI algorithm will reach a point where the programmer is unable to defeat his own AI. For this reason it should be encouraging to the AI programmer if, at the early stages of AI competition, there are more losses than expected. If you continue to study the game and improve your heuristics for the next competition, then the knowledge gained from those losses will certainly pay off by the time the final AI competition rolls around.

Our approach began with a standard game tree. From an initial configuration of pieces on the board, we branched off to all possible next moves. For Pentago, this resulted in an unwieldy tree structure whose depth and branching factor was too large to compute in a reasonable amount of time. Therefore, tree traversal was limited to a depth which examined only a handful of moves ahead.

Initially, we attempted Alpha-Beta pruning alongside a min-max algorithm for optimization. Heuristic evaluation was the main challenge for this implementation and was never quite hammered out effectively until close to the end of the project. Unfortunately this algorithm was slower than other implementations already made at that point and not as effective in strategy, so we kept with other well tested and trusted algorithms and soldiered on to test and improve those AI implementations.

During the interim while Alpha-Beta heuristics were put on the backburner, another attempt to building the AI was to try a MonteCarlo Algorithm. This was a slightly easier algorithm to do, and would play very decently in initial tests against human players as well as the AI's which were already made at that point. Around this time we also found that MonteCarlo algorithms can be easily parallelized and so this optimization was added, and the game played very successfully for a time, but was later soundly beat out by advances in both the heuristic solution for the Alpha-Beta algorithm as well as the final AI implementation which we used for our final solution.

Our final stab at an AI was a simple negamax algorithm, but with a sophisticated heuristic evaluation that made use of precomputed game states. Weights assigned to various configurations of pieces yielded a fast algorithm with accurate heuristic evaluations. From this point we tinkered with the weights by pitting numerous versions of this AI against each other in several games and constantly selecting the best one from the winner pool of games.

The AI and the networking parts of the game are probably the most technically challenging parts of a project like this. It is imperative to try new things and *test, test, test*. Do a lot of research into AI algorithms and study the game closely to find the most useful heuristics. Every little thing counts; changing one thing to be static might make all the difference, so attention to the finer details is important if not vital to coding the best AI. Lastly, take pride in the work you put into the AI and be a positive challenger for the other teams.

Barrett:

Our team agreed at the beginning of the semester that anyone who wanted to make an AI could, and that we would keep the most successful one as our tournament AI. David and I made several different AI players throughout the semester and had a lot of fun pitting them against each other. I think this was a great way to approach the AI portion of the project, because every AI that was made was helping to improve the others.

The final AI (FinalPlayer.h) was a negamax approach using a lookup table of about 1800 precomputed board configurations. To create the table, I first hand-wrote the bitboards for every possible win condition. I also hand wrote bitboards for every possible rotate-to-win situation. The rotate-to-win was an important addition and was designed to combat the Despicable Mentago team's AI, because our old AI was consistently getting cornered into a

situation where there was more than one way to rotate to win. The rotate-to-win table effectively simulates another partial level deep into the tree.

When I finished with those tables, I wrote a program to loop through each configuration and generate a C++ file that would represent a static map where every permutation of each configuration is the key, and the value is the Hamming weight (or pop count) of the configuration. We then copied this large map code into our original code, which was used in the board evaluation function. The boards in a level are weighted by orders of magnitude (using a long double) depending on the permutations matched in the tree and their corresponding Hamming weights. You can see this in the function FinalPlayer::evaluateBitboard.

NETWORKING

Barrett:

Russell and I pair-programmed the networking. I believe we each spent about 60 hours on it. I've never had the networking class, so this was a big learning experience for me. Russell was able to answer a lot of the questions I had about the networking. We intended to be able to reconnect the after a disconnection, and we had the architecture to do it, but we decided that the amount of time it would take to fully implement and test the feature wouldn't be worth the payoff in the end. So a lot of what you see in our networking code (stacks, etc) is overkill for what the code actually does.

We did all of our networking with UDP, using Qt's QUdpSocket objects. Since UDP isn't guaranteed to transmit, we created a Barrager class that sends everything 6 times in rapid succession. This was probably massive overkill for a modern LAN router, in retrospect. We used strictly UDP because I didn't want to have to learn two protocols, but in retrospect, I wish we would've used TCP like how the Pantheon Pentago team uses it.

Anything that goes across the network is in a Transaction. A Transaction object contains a union that can hold many different kinds of transaction data. The Transaction object has an enum that determines which kind of data is in the data union.

We have a strictly P2P network structure that doesn't have a host. The network lobby is dynamic so that when anyone enters, they become visible immediately. When they are in the middle of a game or in the middle of a challenge transaction, they are unable to be challenged and instead are marked as "busy." The network lobby took the most work to finish, and we wish we could've put more time into it. This is done by broadcasting "announces" (a kind of Transaction) every few milliseconds, and everyone listens for them to stop/start to tell whether they are in the lobby. There is also an isBusy flag (added very late in development) that is inside an announce which is toggled accordingly.

As for the game transactions, a turn is simply included in the Transaction and each instance's game controller registers the moves in parallel. Each instance figures out on its own that a game is over.

Russell:

Dividing tasks between group members can be good at times. It seems like more work gets done, but is it quality work? Our group used a lot of pair programming for tasks. I'm not

sure that it was ever intentional. We never actually used or acknowledged the terms 'agile' or 'pair-programming,' but that's what we needed to get our project done.

I feel like networking is a good task to pair-program. Truth be told, I did very little programming for networking. I was there, and I helped test it to make it work. I shared my knowledge and lent a hand when I could. I helped to bounce ideas and catch minor syntax and logic errors. Barrett and I worked on the networking together through pair-programming. I really enjoy pairing and would suggest that you pair up for some tasks during your capstone course. It helps keep focus on the task at hand and makes sure that code is being written in a well-designed way when two people are working on it at the same time.

HELP

David:

For the game of Pentago, the rules are fairly basic and don't need a lot of explanation. For this reason, it was not a major concern for our team to dedicate much time into developing the help portion of our game. Simply put, we built a few demonstrative game boards and interaction graphics and broke down the process of playing the game into concise language in order to deliver to the players of our game a short and simple explanation of the game.

This part of our game was not considered in development of the game core. Therefore it was put on the proverbial backburner until after UIP and was hurriedly integrated into the game. If I could give some advice as regards the help portion of the project it would be to consider it in design, even if just conceptually. It is important to not underestimate the help.

Barrett:

Besides David's help screens, there were several other considerations made during UI development to facilitate the gameplay process unobtrusively. To list a few:

-Hiding the rotation buttons until they are allowed to be clicked

-Hover properties on everything

-Using beta testing to determine the placement of buttons

-Making the "How To Play" button pulsating so it can't be missed

-using a blinking turn indicator light to always show whose turn it is

INSTALLATION

Barrett:

I wrote the installer using NSIS. I started with an example on the NSIS website and extended it as needed. To build the installer, I needed to have all the MinGW and Qt .dlls in the same folder as the NSIS script (the .nsi file) and the FairytaleFactoryFantastica executable. Do not try to statically link everything so that you don't have to write an installer. It's bad practice, a legal gray area, and difficult to achieve in the first place. I wasted a weekend trying to get a static build of 64bit Qt and MinGW just so I could build statically and not have to write an installer. The install script took about an hour to make. I never had to do anything else to it after I finished it and it worked. That said, get it done *early*. Get a system down that you can feel confident about. Obviously, don't forget to your program build in release mode and to set whatever optimization flags you need (we went with -Ofast, but be careful because that lets the compiler deviate quite a bit from the standard). Use the program called DependencyWalker if you're having trouble figuring out which .dll files to package with your distribution.

I'll use this opportunity to talk about the qresource system as well. In a Qt project, you have the option to either leave all of your media files and qml files outside the project, or you can use the qresource system to literally compile them into your executable. We started out without qresource, but added it later. We really regretted using the qresource system. It's very nice for a lot of things but is unnecessary for an app this size. It pushed our compilation time to over 5 minutes by the end of the semester. A development motif of ours was swaying around in our rolly chairs waiting for the build to finish, looking around the room in boredom. Trust me, you don't want to be in that situation when it's crunch time. You constantly have to rebuild when developing and using the qresource system, a program is literally parsing the bytes of every single image, audio, qml, whatever file and translating those bytes to *static C++ arrays in code, to be compiled by your ordinary C++ compiler*. This is great for portability, good for runtime performance (depending on several factors), but horrible for build times. And the qresource system is still buggy, which is why we had to maintain multiple .qrc files (that in itself was the source of several bugs due to tyopographical errors). It's a really cool system, but I strongly recommend that you don't use it for this project. And the NSIS script could've easily taken care of all those files.

AUDIO

David:

Sounds in the game were a fairly easy addition, but nonetheless invaluable to creating the desired user experience in our game. There are a lot of royalty free sounds available online, and they are perfectly suited for a game project like ours. One piece of advice for those who may be ethically conscious or simply polite: keep records of licensing and sources in case you need the information to give credit. Audacity was a good tool to use when our sound files needed some simple tailoring.

From a technical standpoint, sounds were a simple task with the Qt framework. Initially, we had implemented a slot/signal relationship between C++ and the QML, but when we discovered we could do sound effects directly in the QML it was a breeze to switch over and let Qt do the work for us. QML is well documented and has readily accessible example code to demonstrate this process, so I cannot stress enough how simple this part of the project was compared to other tasks. One thing which was a hassle in sound development was the lack of consideration in the design phase. Since our GUI code was built without consideration for sound, there were far too many checks for "mute" flag states or awkwardly placed blocks of sound effect code which just made the project appear patched together.

Intro Video

Stephanie:

Our intro video took about one and a half weeks to complete, with several of those days spend researching how to make it work. We were originally going to make a video using Windows Movie Maker by creating multiple sets of frames, but when we began to research how to do that in QML, it seemed entirely too complicated. QML has two ways to play video, and one of the ways apparently still had some bugs, while the other way involved finding and downloading necessary libraries which were dependent upon which type of video file you used. Our team was unable to successfully get it working on both Mac and Windows platforms, but the team Pantheon reportedly was able to get it working for their game.

Instead, we decided to animate the intro video in the code itself. The intro screen was made visible once the splash screen had either run its course or been skipped. A timer was triggered once the intro screen was made visible, and it counted down from ten seconds, at which point it turned the invisibility off and made the next screen visible. Once the intro screen was visible, we animated our graphics, turning the visibility on and off and moving pieces around using the built-in animation functionality offered by Qt. Each individual animation was triggered used timers similar to the ten second one. They were put on a loop to continue being triggered until the ten second timer was done. In order to be able to skip video, we set a listening element to listen for a key being pressed, and if so called the same functionality that was called when the ten second timer triggered. The sound for the intro screen was done using the Sound Effect element offered by Qt. It was turned on whenever the intro screen was set to visible and turned off whenever the visibility was set to invisible.